Lecture 26

CIS 341: COMPILERS

Announcements

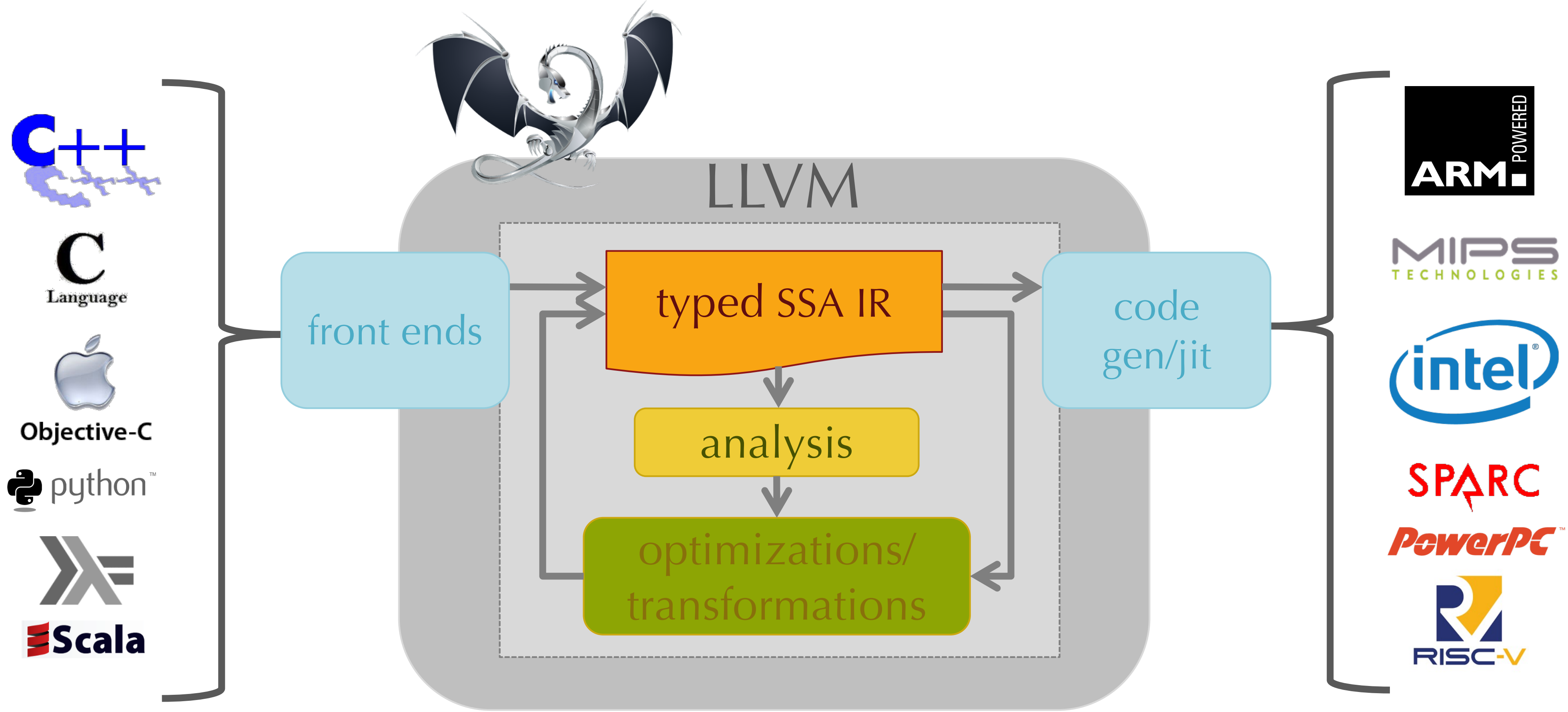
- Final Exam:
  - LRSM AUD
  - Monday, May 2nd noon - 2:00pm

- Current Plan / My Preference: ***In Person***
  - Unless University policy prohibits in person exams, this is the default
  - If you have serious concerns about taking the exam in person, I will make accommodations

COMPILER VERIFICATION

LLVM Compiler Infrastructure

[Lattner et al.]

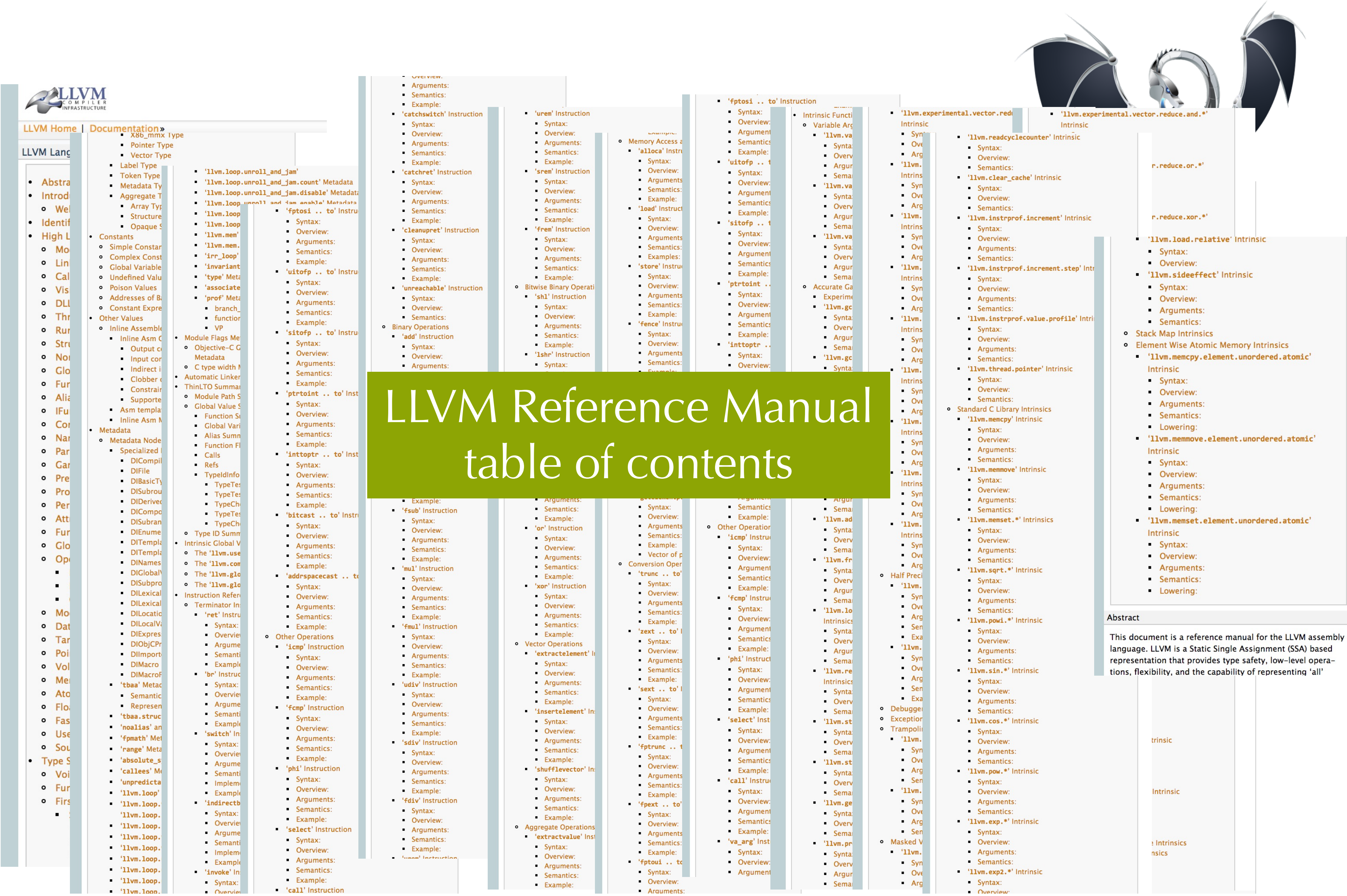


Other LLVM IR Features

- C-style data values
  - ints, structs, arrays, pointers, vectors
- Type system
  - used for layout/alignment/padding
- Relaxed-memory concurrency primitives
- Intrinsics
  - extend the language malloc, bitvectors, etc.
- Transformations & Optimizations

Make targeting LLVM IR easy and attractive for developers!

But… it's complex



One Example: `undef`

The **`undef`** "value" represents an arbitrary, but indeterminate bit pattern for any type.

Used for:
- uninitialized registers
- reads from volatile memory
- results of some underspecified operations

What is the value of **%y** after running the following?

```
%x = or i8 undef, 1
%y = xor i8 %x, %x
```

One plausible answer: 0

Not LLVM's semantics!

(LLVM is more liberal to permit more aggressive optimizations)

Partially defined values are interpreted *nondeterministically* as sets of possible values:

```
%x = or i8 undef, 1
%y = xor i8 %x, %x
```

```
⟦i8 undef⟧ = {0,…,255}
⟦i8 1⟧ = {1}
⟦%x⟧ = {a or b | a∈⟦i8 undef⟧, b ∈⟦1⟧}
     = {1,3,5,…,255}
⟦%y⟧ = {a xor b | a∈⟦%x⟧, b∈⟦%x⟧}
     = {0,2,4,…,254}
```

Interactions with Optimizations

Consider:

```
%y = mul i8 %x, 2
```

```
⟦%x⟧  = ⟦i8 undef⟧
      = {0,1,2,3,4,5,…,255}
⟦%y⟧ = {a mul 2 | a∈⟦%x⟧}
      = {0,2,4,…,254}
```

versus:

```
%y = add i8 %x, %x
```

```
⟦%x⟧  = ⟦i8 undef⟧
      = {0,1,2,3,4,5,…,255}
⟦%y⟧ = {a + b | a∈⟦%x⟧,
b∈⟦%x⟧}
      = {0,1,2,3,4,…,255}
```

≠

Interactions with Optimizations

Consider:

```
%y = mul i8 %x, 2
```

versus:

```
%y = add i8 %x, %x
```

Upshot: if **`%x`** is **`undef`**, we can't optimize **`mul`** to **`add`** (or vice versa)!

What's the problem?

Bug List: (12 of 435) First Last Prev Next Show last search results

Bug 33165 - Simpify* cannot distribute instructions for simplification due to undef

Davide Italiano 2017-05-25 08:55:40 PDT Comment 6

Davide Italiano 2017-05-25 09:05:26 PDT Comme

(unless we want to give up on some undef transformations, and special case sele

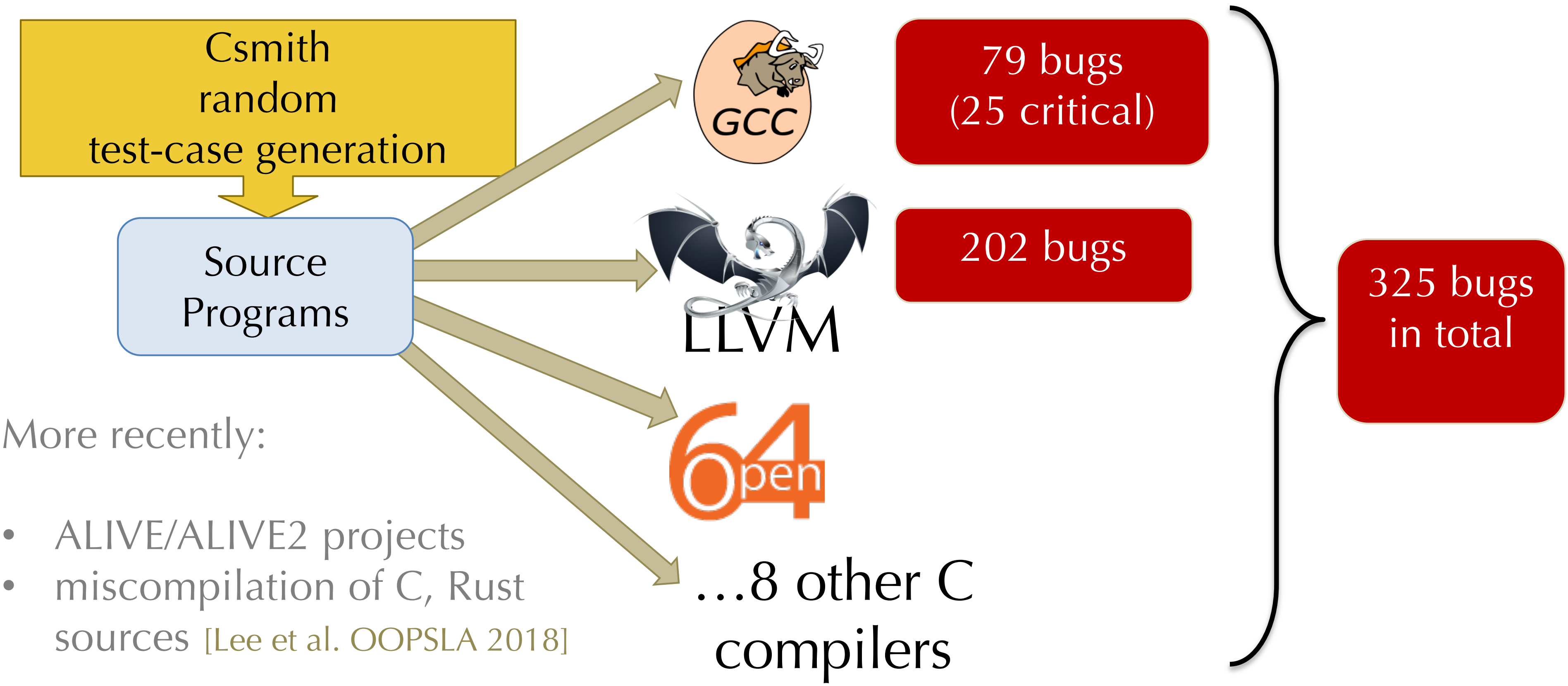
John Regehr 2017-05-25 09:09:24 PDT Comm

Yes, this is one of those test cases. There are so many optimization failures
Nuno has been automatically filtering out classes of mistranslation that are
to be hard to fix but I guess he decided to take a closer look at some of the

Soon I'll be able to include branches/phis in these test cases, but only forw
branches due to a limitation in Alive.

Compiler Bugs

[Regehr's group: Yang et al. PLDI 2011]



More recently:

- ALIVE/ALIVE2 projects
- miscompilation of C, Rust sources [Lee et al. OOPSLA 2018]

LLVM is hard to trust
(especially for critical code)

What can we do about it?

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound programming languages tools
  - "Formal" verification

Less "formal": Techniques may miss problems in programs

This isn't a tradeoff… all of these methods should be used.
Even "formal" methods can have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth. "Beware of bugs in the above code; I have only proved it correct, not tried it."

More "formal": eliminate with certainty as many problems as possible.

Goal: Verified Software Correctness

- Social
  - Code reviews
  - Extreme/Pair programmi
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - “lint” tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound programming languages tools
  - “Formal” verification

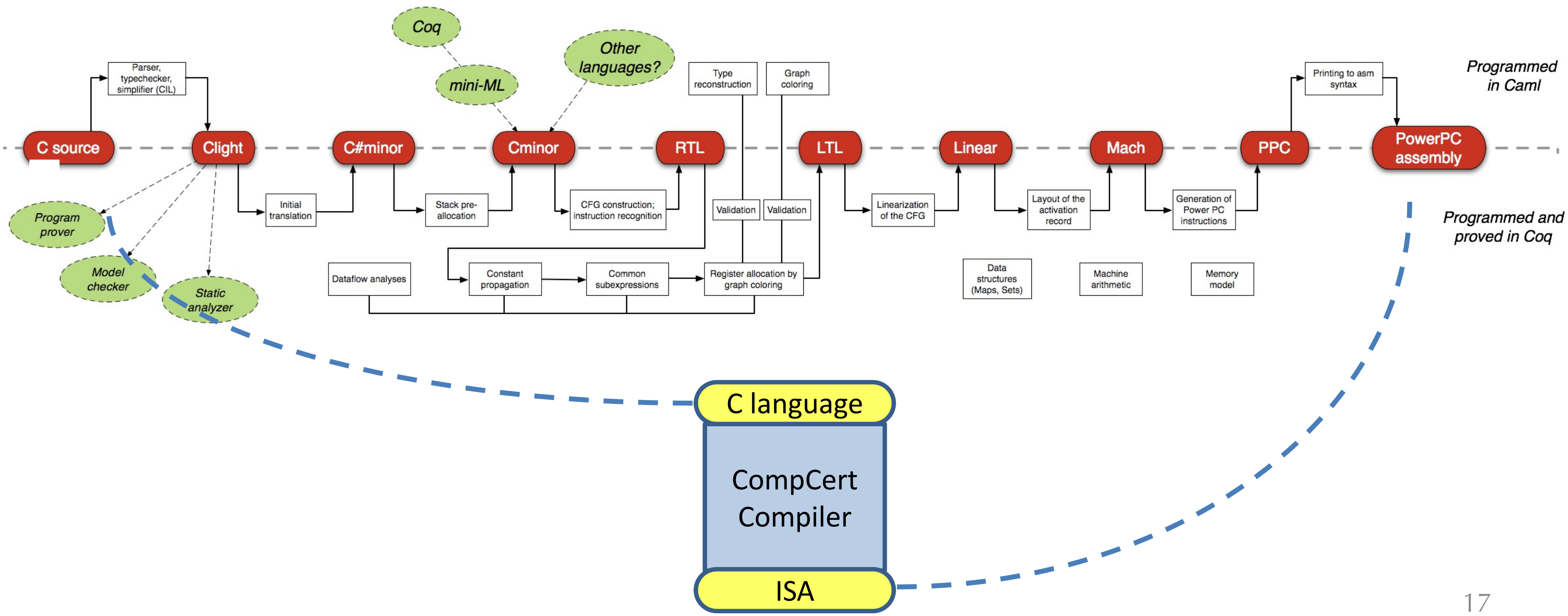
Q: How can we move the needle towards mathematical software correctness properties?

Taking advantage of advances in computer science:
- Moore's law
- improved programming languages & theoretical understanding
- better tools: interactive theorem provers

CompCert – A Verified C Compiler

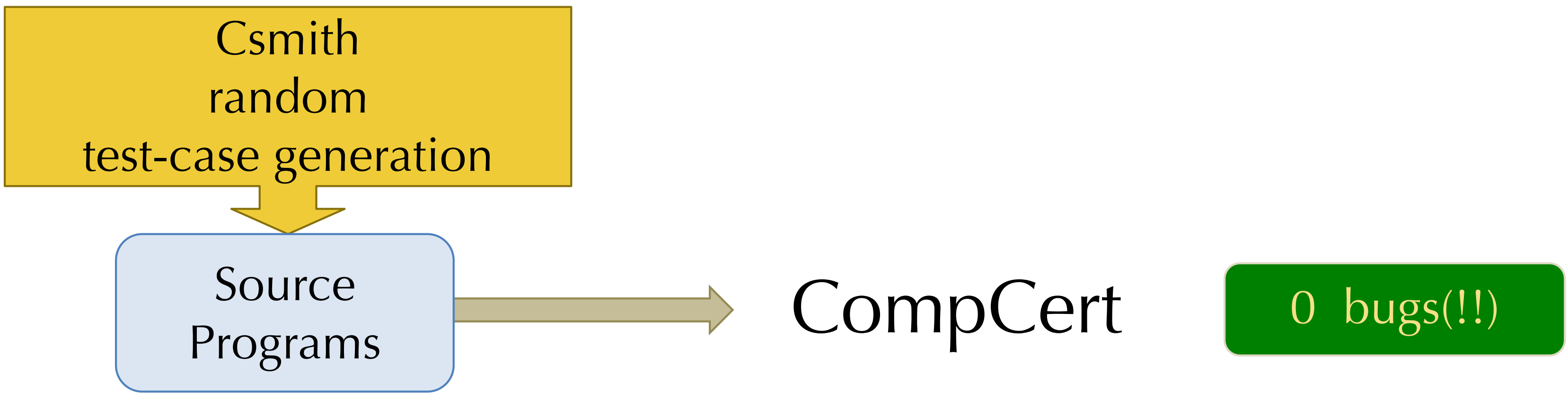
Xavier Leroy
INRIA

Optimizing C Compiler,
proved correct end-to-end
with machine-checked proof in Coq



Csmith on CompCert?

[Yang et al. PLDI 2011]



Verification Works!

"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested ***for which Csmith cannot find wrong-code errors***. This is not for lack of trying: we have devoted about six CPU-years to the task. ***The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.***"

– Regehr et. al 2011

Our Approach: Formal Verification

Interactive theorem proving in Coq

- not model checking / SMT
- human-in-the-loop

Using Coq ***is*** functional programming

…but some of your programs *are* proofs

⇒ proof engineering

CERTIKOS

Verified Software Toolchain

QuickChick

deep spec

Core Spec

Vellvm verified LLVM

Kami

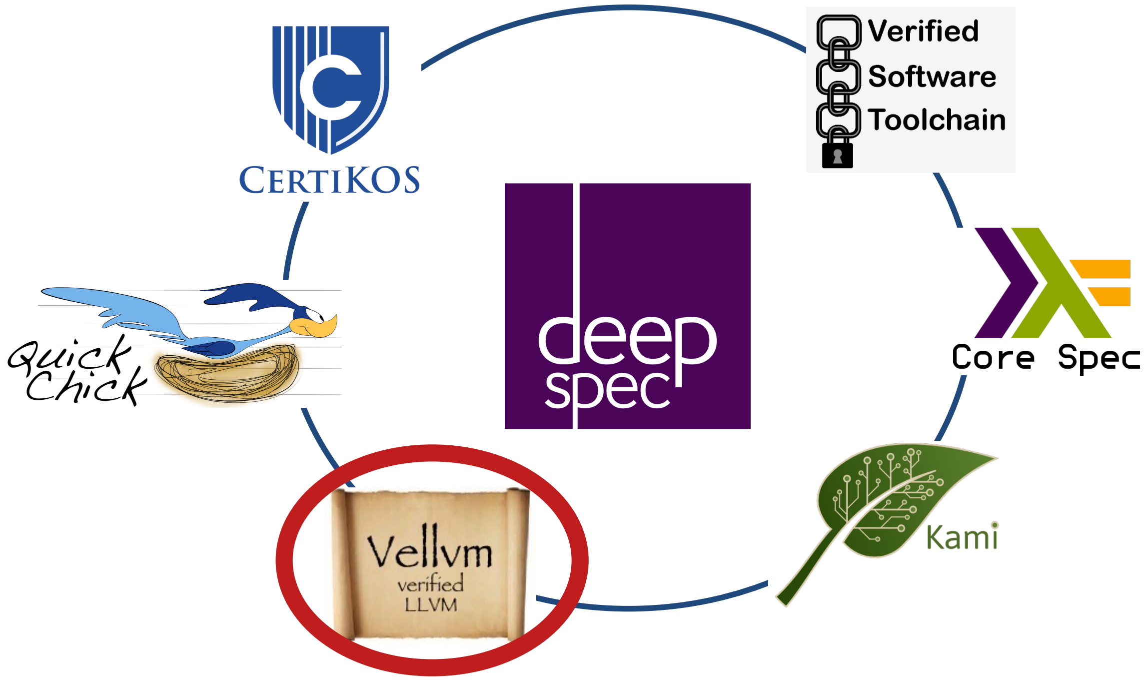
deepspec.org

Deep Specifications

[deepspec.org]

- ***Rich*** – expressive description
- ***Formal*** – mathematical, machine-checked
- ***2-Sided*** – tested from both sides
- ***Live*** – connected to real, executable code

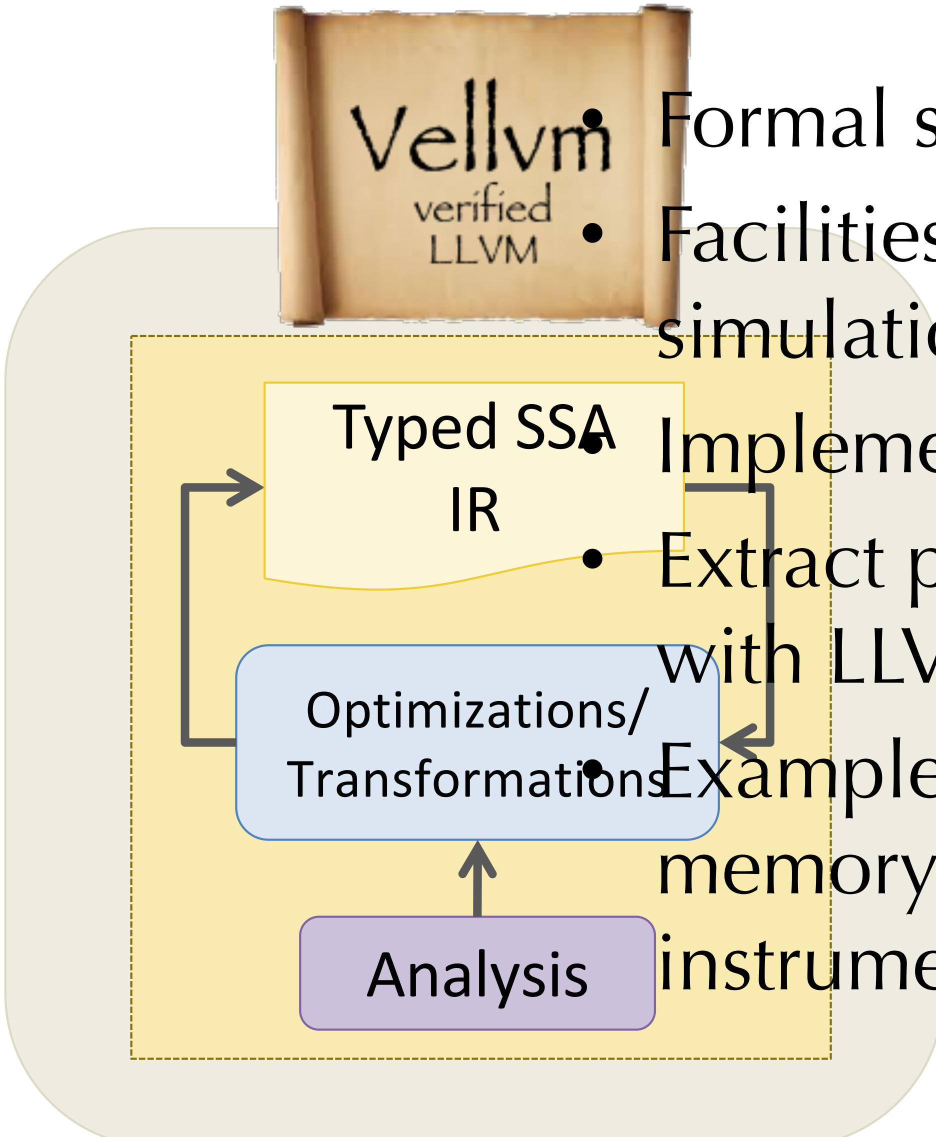
Goal: Advance the reliability, safety, security, and cost-effectiveness of software (and hardware).



deepspec.org

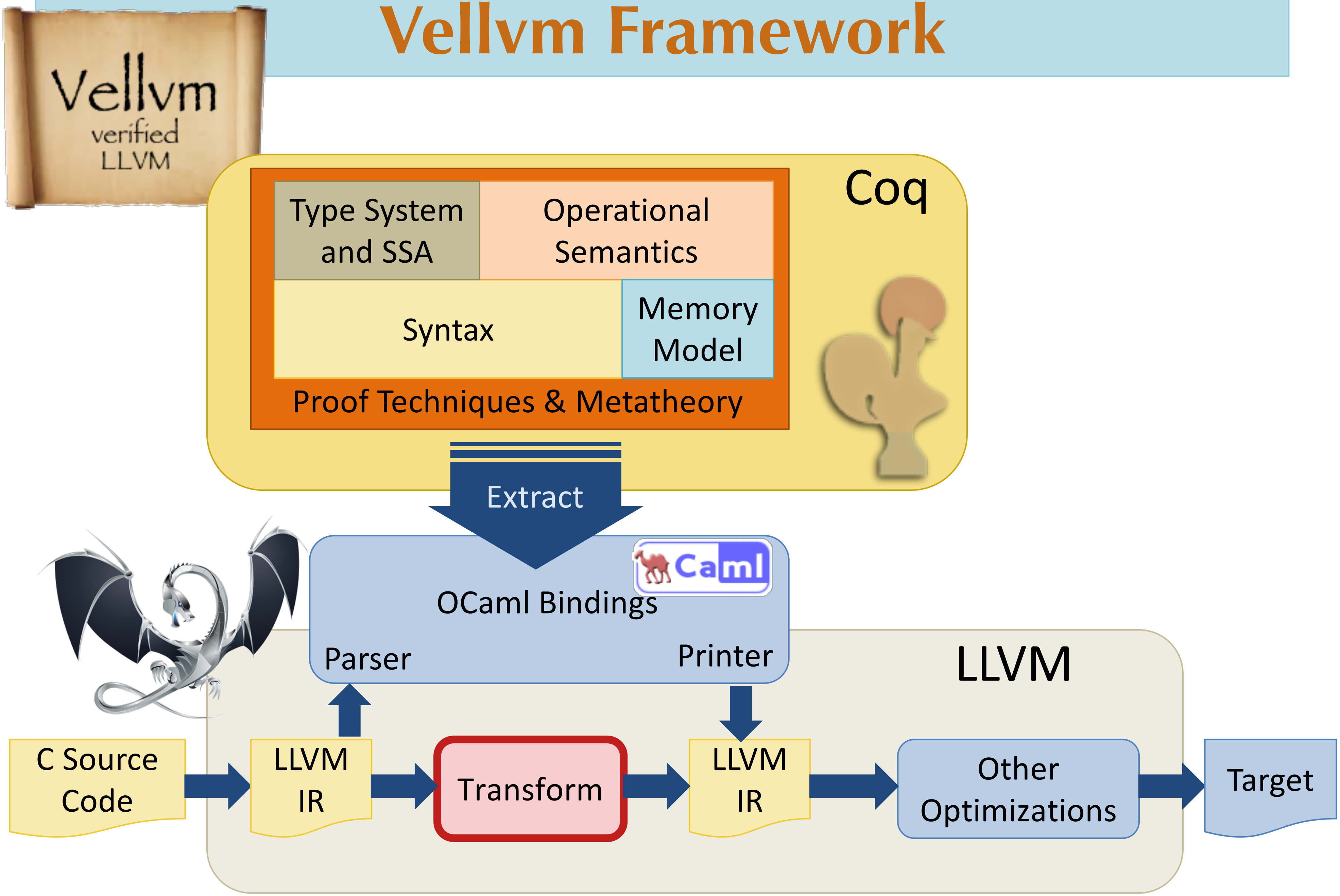
The Vellvm Project

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013, Zackowski, et al. ICFP2021]

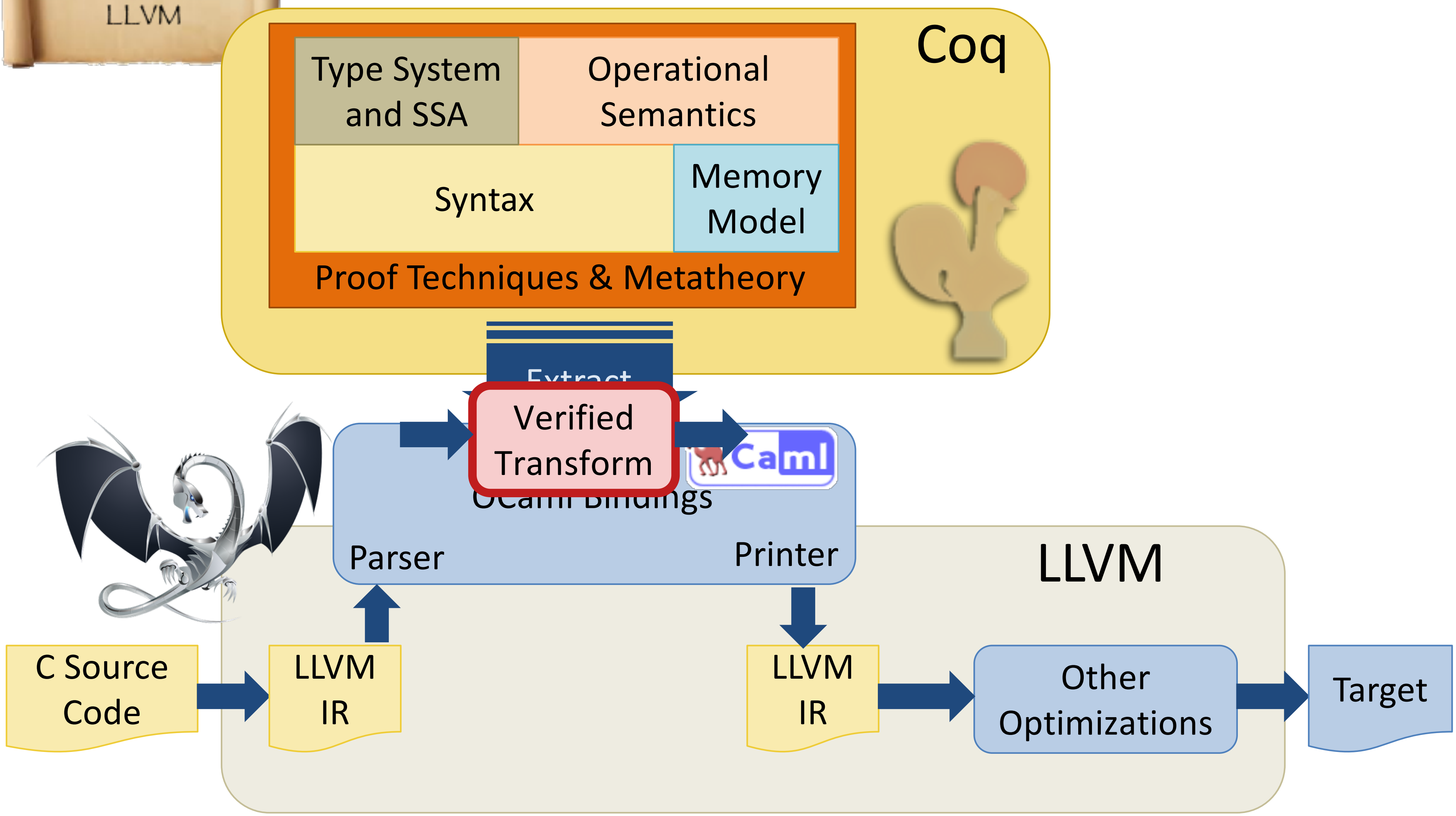


- Formal semantics
- Facilities for creating simulation proofs
- Implemented in Coq
- Extract passes for use with LLVM compiler
- Example: verified memory safety instrumentation

Vellvm Framework



Vellvm Framework



Writing Interpreters in Coq

Galina (Coq's Language)

- rich, dependent type system
- pure, total functional language

How do we write the interpretation function?

```
Inductive exp : Set :=
| EXP_Ident   (id:ident)
| EXP_Integer (x:int)
| EX
| ..
```

```
Inductive instr : Set :=
| INSTR_Op   (op:exp)                               (* IN
| INSTR_Call (fn:texp) (args:list texp)             (* CO
| INSTR_
| INSTR_
| INSTR_
```

```
Definition code := list (instr_id * instr).

Record block : Set :=
  mk_block
    {
      blk_id    : block_id;
      blk_phis  : list (local_id * phi);
      blk_code  : code;
      blk_term  : instr_id * terminator;
    }.
```

Datatypes for Abstract Syntax

LLMV Memory Model (simplified)

Describes the interface for "observations" of LLVM IR programs.

```
(* IO interactions for the LLVM IR *)
Inductive IO : Type -> Type :=
| Alloca : ∀ (t:dtyp), (IO dvalue)
| Load   : ∀ (t:dtyp) (a:dvalue), (IO dvalue)
| Store  : ∀ (a:dvalue) (v:dvalue), (IO unit)
| GEP    : ∀ (t:dtyp) (v:dvalue) (vs:list dvalue), (IO dvalue)
| ItoP   : ∀ (i:dvalue), (IO dvalue)
| PtoI   : ∀ (a:dvalue), (IO dvalue)
| Call   : ∀ (f:string) (args:list dvalue), (IO dvalue)
.
```

output values of the Call event

type of the result provided by the environment

LLVM Interpreter in Coq

```
Definition step (s:state) : LLVMTrace result
  let '(g, pc, e, k) := s in

  do cmd ← trywith ("CFG has no instruction at " ++ string_of
  match cmd with
  | Term (TERM_Ret (t, op)) ⇒
    'dv ← eval_exp (Some (eval_typ t)) op;
      match k with
      | [] ⇒ halt dv
      | (KRet e' id p') :: k' ⇒ cont (g, p', add_env id dv e', k'
      | _ ⇒ raise_p pc "IMPOSSIBLE: Ret op in non-return configura
      end

  | Inst insn ⇒  (* instruction *)
    do pc_next ← trywith "no fallthrough instruction" (incr_pc CFG pc);
    match (pt pc), insn  with

      | IId id, INSTR_Op op ⇒
         'dv ← eval_op g e op;
          cont (g, pc_next, add_env id dv e, k)

      | IId id, INSTR_Alloca t _ _ ⇒
        Trace.Vis (Alloca (eval_typ t))
                  (λ (a:dvalue) ⇒ cont (g, pc_next, add_env id a e, k))

      | IId id, INSTR_Load _ t (u,ptr) _ ⇒
        'dv ← eval_exp (Some (eval_typ u)) ptr;
          Trace.Vis (Load (eval_typ t) dv)
                    (λ dv ⇒ cont (g, pc_next, add_env id dv e, k))
```

interpreter returns
an interaction tree
with "LLVM" effects.
LLVMTrace := itree IO

Extract to executable
interpreter (Ocaml).

The interpreter
"calls out" to the memory
model by generating
visible effects…

Interactive Theorem Proving

In Coq, one can state Lemmas just as easily as any other kind of function.

```
Theorem block_fusion_cfg_correct :
  ∀ (G : cfg dtyp),
    wf_cfg G →
    ⟦ G ⟧cfg ≈ ⟦ block_fusion_cfg G ⟧cfg.
Proof.
  intros G [WF1 WF2].
  unfold denote_cfg.
  simpl bind.
  unfold block_fusion_cfg.
  destruct (b  k_fusion G.(blks))  :EQ.
  - break_match_goa  reflexivit
    simpl.
    apply Bool.orb_false_elim in
    unfold Eqv.eqv_dec in *.
```

You can prove those lemmas interactively. Coq checks each step as you do it.

Comparing Behaviors

- Consider two programs P1 and P2 possibly in different languages.
  - e.g., P1 is an Oat program, P2 is its compilation to LL

- The semantics of the languages associate to each program a set of observable behaviors:

$$\mathfrak{B}(P) \quad \text{and} \quad \mathfrak{B}(P')$$

- Note: $|\mathfrak{B}(P)| = 1$ if P is deterministic, > 1 otherwise

What is Observable?

- For C-like languages:

```
observable behavior ::=
    | terminates(st)        (i.e. observe the final state)
    | diverges
    | goeswrong
```

- For pure functional languages:

```
observable behavior ::=
    | terminates(v)         (i.e. observe the final value)
    | diverges
    | goeswrong
```

What about I/O?

- Add a *trace* of input-output events performed:

	t	::=	[]	|	e :: t	(finite traces)
coind.	T	::=	[]	|	e :: T	(finite and infinite traces)

observable behavior ::=
| terminates(t, st) (end in state st after trace t)
| diverges(T) (loop, producing trace T)
| goeswrong(t)

Examples

- P1:
  `print(1);` / st ⇒ terminates(out(1)::[],st)

- P2:
  `print(1); print(2);` / st
  ⇒ terminates(out(1)::out(2)::[],st)

- P3:
  `WHILE true DO print(1) END` / st
  ⇒ diverges(out(1)::out(1)::…)

- So $\mathfrak{B}(P1) \neq \mathfrak{B}(P2) \neq \mathfrak{B}(P3)$

Bisimulation

- Two programs P1 and P2 are bisimilar whenever:

$$\mathfrak{B}(P1) = \mathfrak{B}(P2)$$

- The two programs are completely indistinguishable.

- But… this is often too strong in practice.

Compilation Reduces Nondeterminism

- Some languages (like C) have underspecified behaviors:
  - Example: order of evaluation of expressions f() + g()
- Concurrent programs often permit nondeterminism
  - Classic optimizations can reduce this nondeterminism
  - Example:

    a := x + 1; b := x + 1 || x := x+1

    vs.

    a := x + 1; b := a || x := x+1
- LLVM explicitly allows nondeterminism:
  - undef values (not part of LLVM lite)
  - see the discussion later

Backward Simulation

- Program P2 can exhibit fewer behaviors than P1:

$$\mathfrak{B}(P1) \supseteq \mathfrak{B}(P2)$$

- All of the behaviors of P2 are permitted by P1, though some of them may have been eliminated.
- Also called ***refinement***.

What about goeswrong?

- Compilers often translate away bad behaviors.

x := 1/y ; x := 42 vs. x := 42

(divide by 0 error) (always terminates)

- Justifications:
  - Compiled program does not "go wrong" because the program type checks or is otherwise formally verified
  - Or just "garbage in/garbage out"

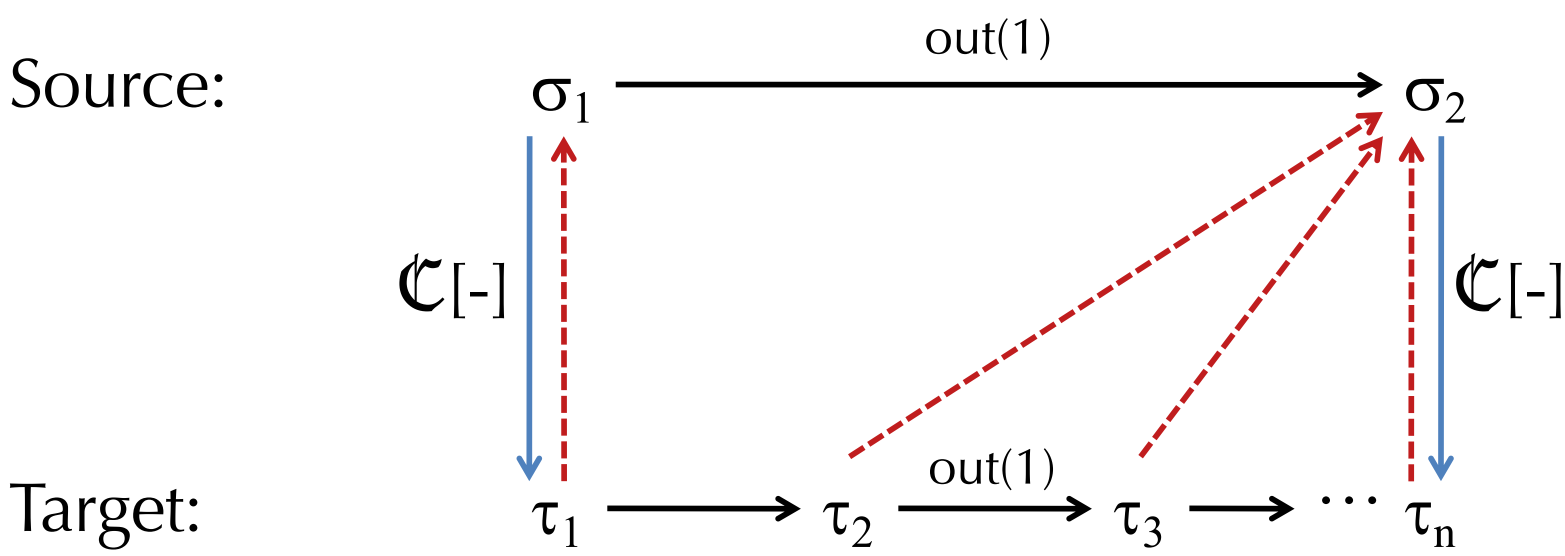
Safe Backwards Simulation

- Only require the compiled program's behaviors to agree if the source program could not go wrong:

$$\text{goeswrong(t)} \notin \mathfrak{B}(\text{P1}) \quad \Rightarrow \quad \mathfrak{B}(\text{P1}) \supseteq \mathfrak{B}(\text{P2})$$

- Idea: let S be the *functional specification* of the program: A set of behaviors not containing goeswrong(t).
  - A program P satisfies the spec if $\mathfrak{B}(\text{P}) \subseteq \text{S}$
- Lemma: If P2 is a safe backwards simulation of P1 and P1 satisfies the spec, then P2 does too.

Building Backward Simulations



Idea: The event trace along a (target) sequence of steps originating from a compiled program must correspond to some source sequence.

Tricky parts:

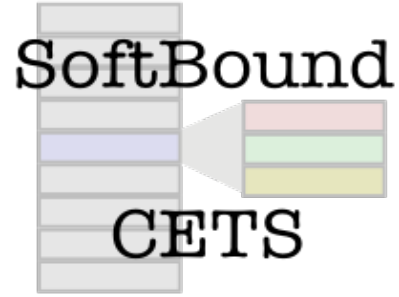
- Must consider all possible target steps
- If the compiler uses many target steps for once source step, we have to invent some way of relating the intermediate states to the source.
  - the compilation function goes the wrong way to help!

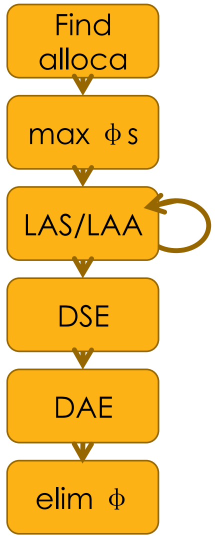
So What?

- Find bugs in the existing LLVM infrastructure
  - thinking hard about corner cases while formalizing is a good way to find real bugs
  - identify inconsistent assumptions on the LLVM compiler
- Automated Tests against other implementations
  - e.g., integrate with Csmith
- Formally validate program transformations
  - is a particular optimization correct?
  - improve confidence in novel program transformations
- Eventually… verify compiler front ends and/or back ends
  - to obtain a fully-verified CompCert-like compiler

VELLVM [Previous Results]

- Verified SoftBound
  - Memory Safety
- Verified mem2reg
  - Register promotion, defined in terms of a stack of "micro-optimizations"
- Verified dominator analysis
  - Cooper-Harvey-Kennedy Algorithm
- Better memory models
  - ptrtoint casts
  - modular formalization





Can it Scale?

- Use of theorem proving to verify "real" software is still considered to be the bleeding edge of research.

- **CompCert** – fully verified C compiler
  Leroy, INRIA
- **Vellvm** – formalized LLVM IR
  Zdancewic, Penn
- **Ynot** – verified DBMS, web services
  Morrisett, Harvard
- **Verified Software Toolchain**
  Appel, Princeton
- **Bedrock** – web programming, packet filters
  Chlipala, MIT
- **CertiKOS** – certified OS kernel
  Shao & Ford, Yale
- **CakeML** – certified compiler
- **SEL4** – certified secure OS microkernel
- **Kami** – verified RISCV architecture
- **DaisyNSF** – verified NFS file system
- …

Formal Methods for Blockchain

Academic Work:
A Survey of Smart Contract Formal Specification and Verification
[Tolmach, et al. 2021]

Uses deep spec results

Where next?

- Proof engineering is still nascent
  - automation, scale, maintenance
  - software engineering++
  - new theory needed: dealing with equality
- Verification is still hard
  - labor intensive, difficult, $$$$
- Deep Specifications
  - what are the principles?
  - compositionality?
- Real-time, cyberphysical,…

What have we learned?

Where else is it applicable?

What next?

COURSE WRAP-UP

Final Exam

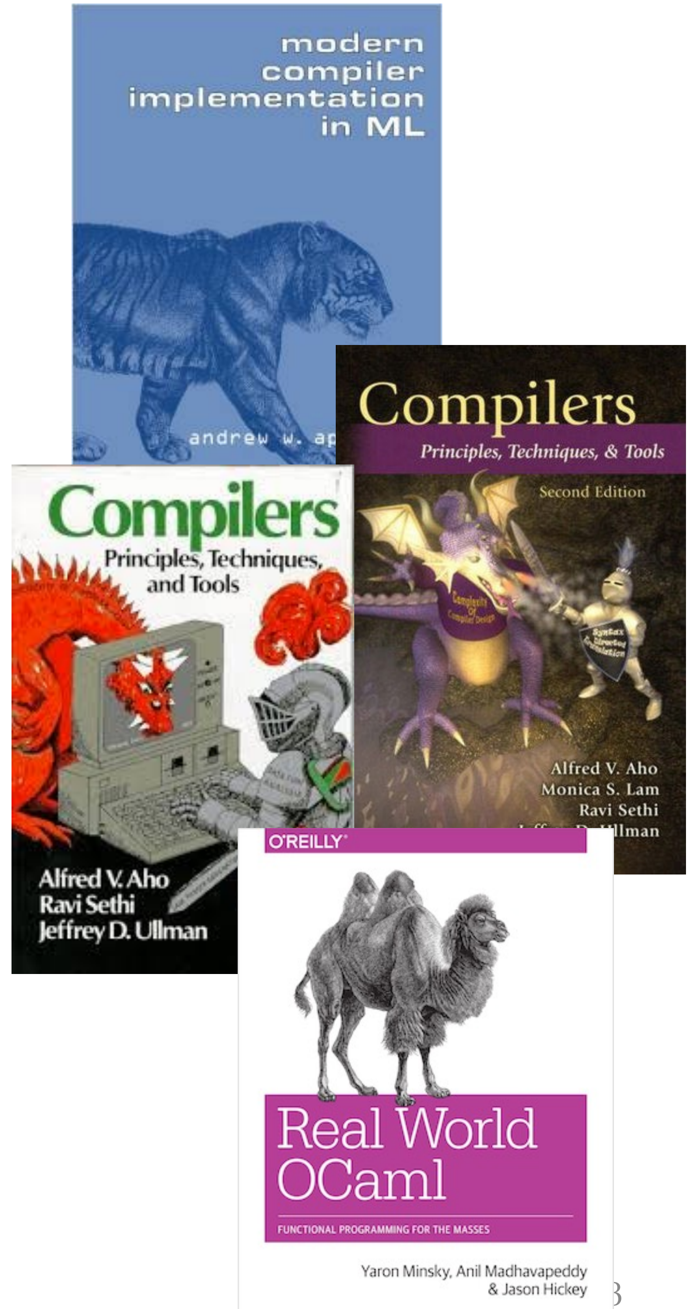
- Will mostly cover material since the midterm
  - Starting from Lecture 14
  - Lambda calculus / closure conversion
  - Scope / Typechecking / Inference Rules
  - Objects, inheritance, types, implementation of dynamic dispatch
    (de-emphasized, since we didn't cover it thoroughly)
  - Basic optimizations
  - Dataflow analysis (forward vs. backward, fixpoint computations, etc.)
    - Liveness
  - Graph-coloring Register Allocation
  - Control flow analysis
    - Loops, dominator trees

- One, letter-sized, double-sided, hand-written "cheat sheet"

Why CIS 341?

- You will learn:
  - Practical applications of theory
  - Parsing
  - How high-level languages are implemented in machine language
  - (A subset of) Intel x86 architecture
  - A deeper understanding of code
  - A little about programming language semantics
  - Functional programming in OCaml
  - How to manipulate complex data structures
  - How to be a better programmer

- Did we meet these goals?



Stuff we didn't Cover

- We skipped stuff at every level…
- Concrete syntax/parsing:
  - Much more to the theory of parsing… LR(*)
  - Good syntax is art, not science!
- Source language features:
  - Exceptions, advanced type systems, type inference, concurrency
- Intermediate languages:
  - Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
- Compilation:
  - Continuation-passing transformation, efficient representations, scalability
- Optimization:
  - Scientific computing, cache optimization, instruction selection/optimization
- Runtime support:
  - memory management, garbage collection

Lexing
Parsing
Disambiguation
Semantic analysis
Translation
Control-flow analysis
Data-flow analysis
Register allocation
Code emission

Compiler Passes

Related Courses

- CIS 500: Software Foundations
  - Prof. Pierce
  - Theoretical course about functional programming, proving program properties, type systems, lambda calculus. Uses the theorem prover Coq.
- CIS 501: Computer Architecture
  - Prof. Devietti
  - 371++: pipelining, caches, VM, superscalar, multicore,…
- CIS 547: Software Analysis
  - Prof. Naik
  - LLVM IR + program analysis
- CIS 552: Advanced Programming
  - Prof. Weirich
  - Advanced functional programming in Haskell, including generic programming, metaprogramming, embedded languages, cool tricks with fancy type systems
- CIS 670: Special topics in programming languages

Where to go from here?

- Conferences (proceedings available on the web):
  - Programming Language Design and Implementation (PLDI)
  - Principles of Programming Langugaes (POPL)
  - Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  - International Conference on Functional Programming (ICFP)
  - European Symposium on Programming (ESOP)
  - …
- Technologies / Open Source Projects
  - Yacc, lex, bison, flex, …
  - LLVM – low level virtual machine
  - Java virtual machine (JVM), Microsoft's Common Language Runtime (CLR)
  - Languages: OCaml, F#, Haskell, Scala, Go, Rust, …?

Where else is this stuff applicable?

- General programming
  - In C/C++, better understanding of how the compiler works can help you generate better code.
  - Ability to read assembly output from compiler
  - Experience with functional programming can give you different ways to think about how to solve a problem
- Writing domain specific languages
  - lex/yacc very useful for little utilities
  - understanding abstract syntax and interpretation
- Understanding hardware/software interface
  - Different devices have different instruction sets, programming models

Thanks!

- To the TAs: Stephen, Lef, and Sumanth
- To *you* for taking the class!
- How can I improve the course?
  - Let me know in course evaluations!