Lecture 25

CIS 4521/5521: COMPILERS

Announcements

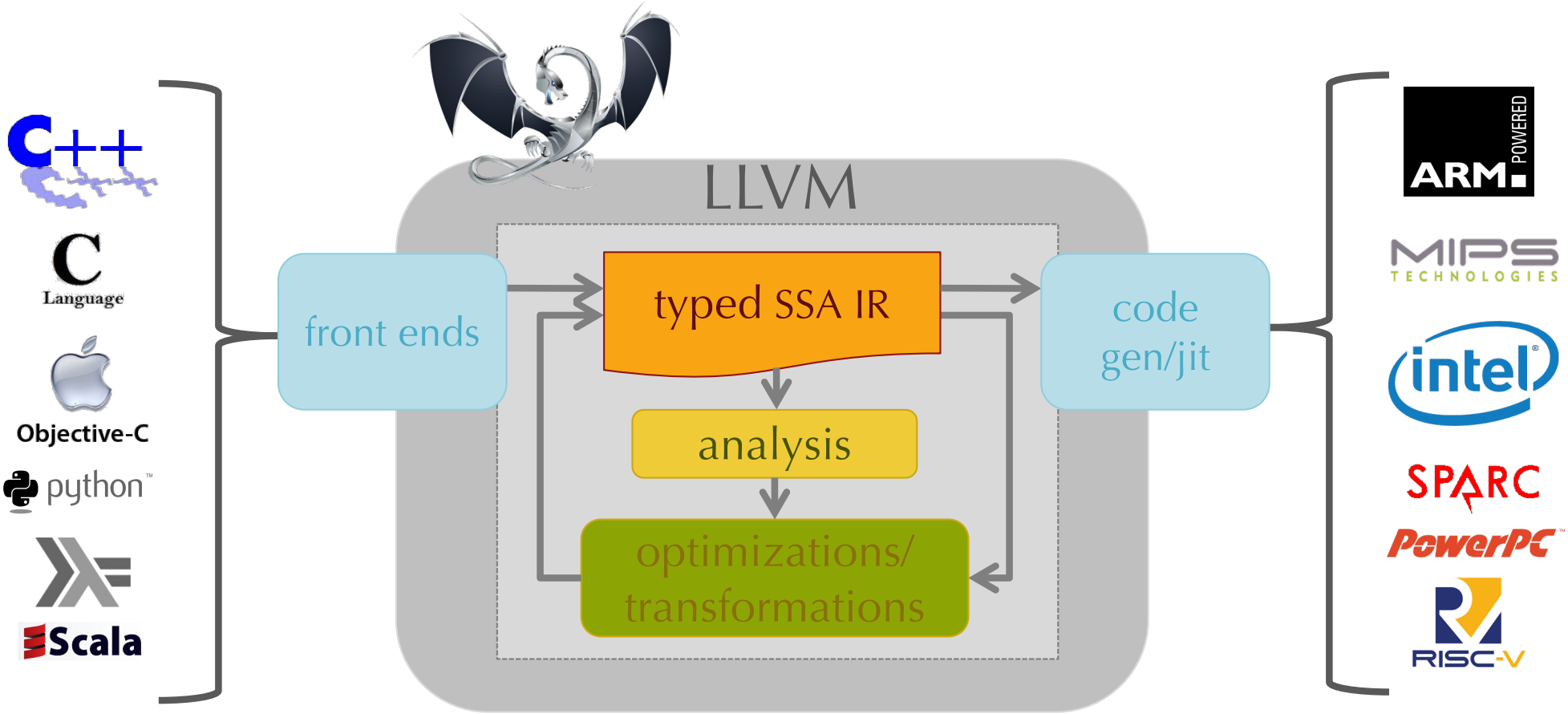
- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Due: **TOMORROW** - Wednesday, April 30th at 10:00pm
  - Posted test case due **tonight** at 10:00pm

- Final Exam:
  - Thursday, May 8th noon - 2:00pm
  - Coverage: emphasizes material since the midterm
  - Cheat sheet: one, hand-written, double-sided, letter-sized page of notes

COMPILER VERIFICATION

LLVM Compiler Infrastructure

[Lattner et al.]

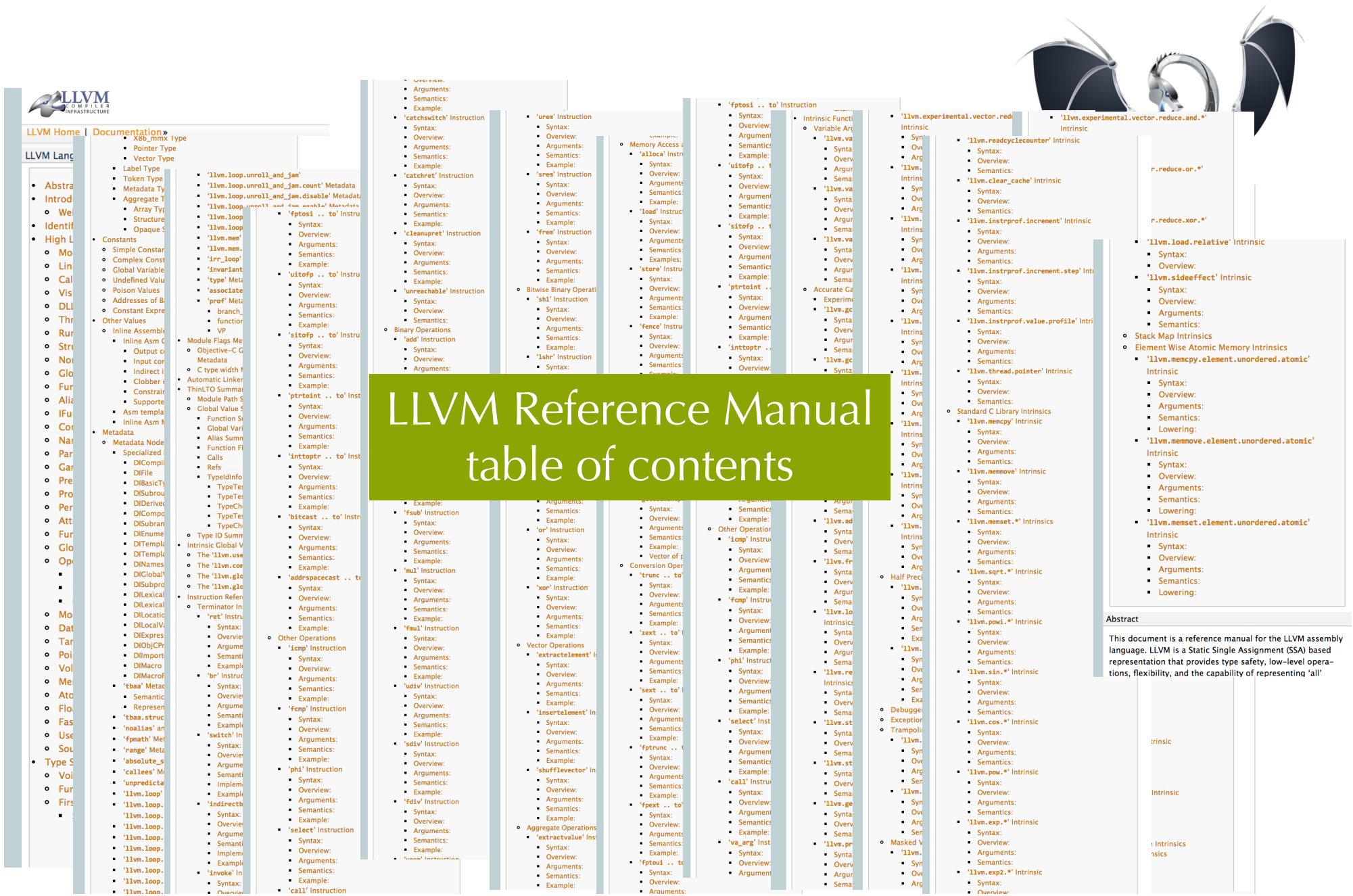


Other LLVM IR Features

- 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
```

versus:

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

```
%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:

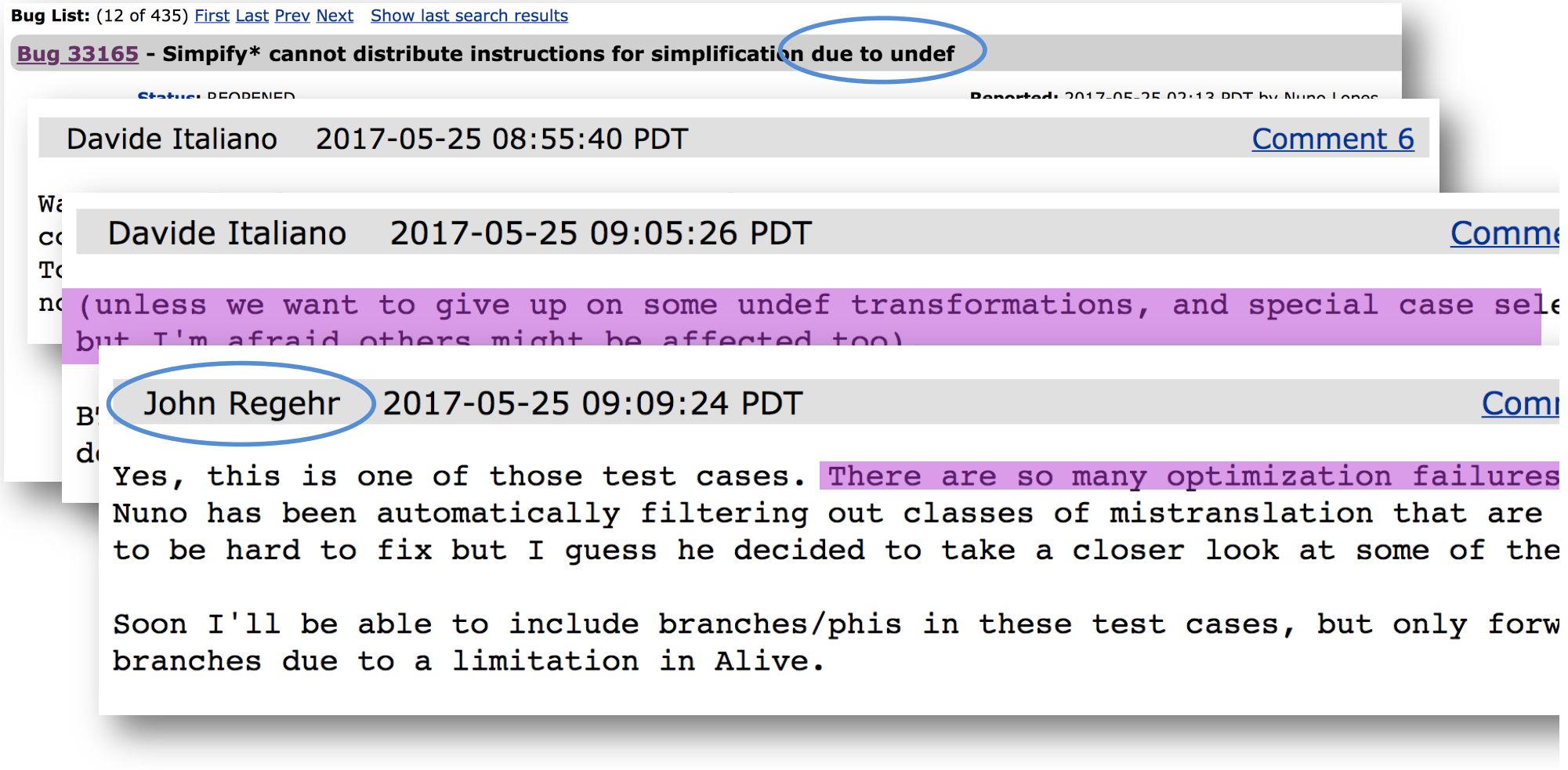
```
%y = mul i8 %x, 2
```

versus:

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

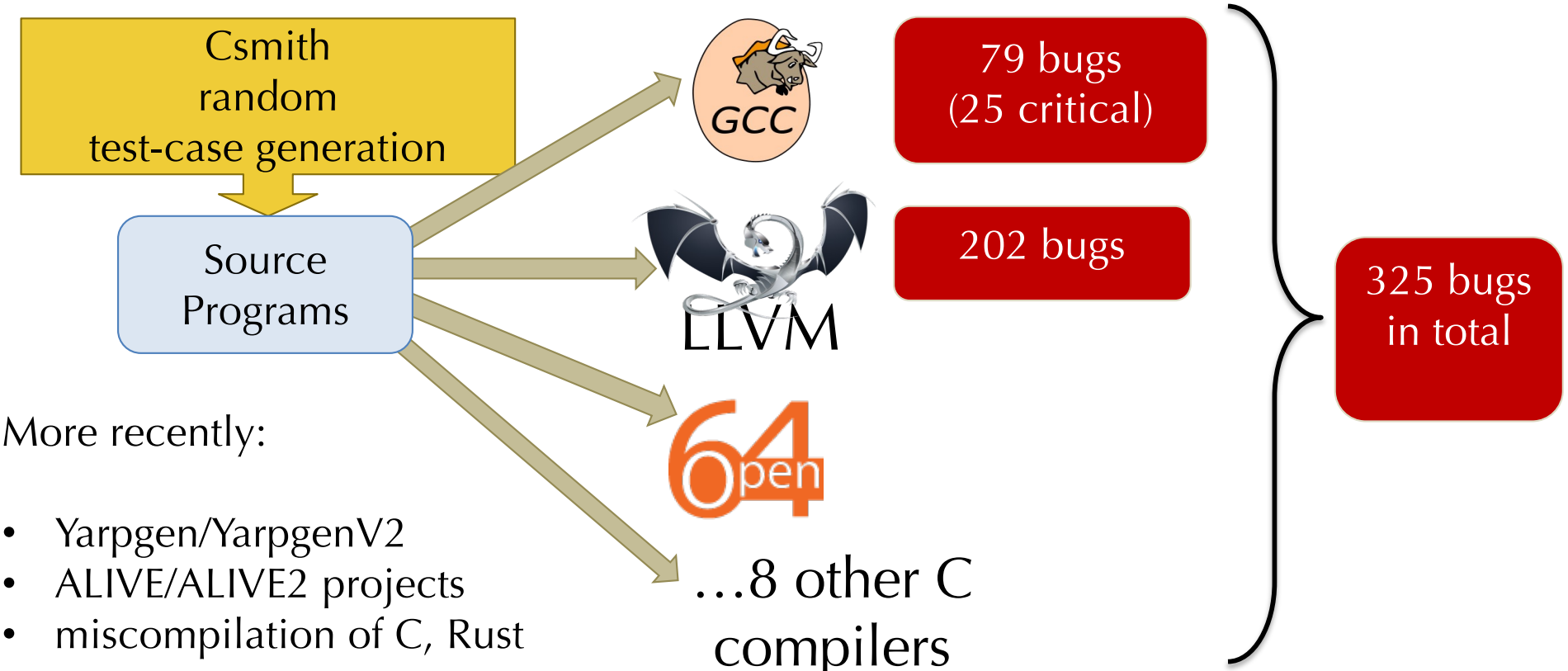
Upshot: if **%x** is **undef**, we can't optimize **mul** to **add** (or vice versa)!

What's the problem?



Compiler Bugs

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



More recently:

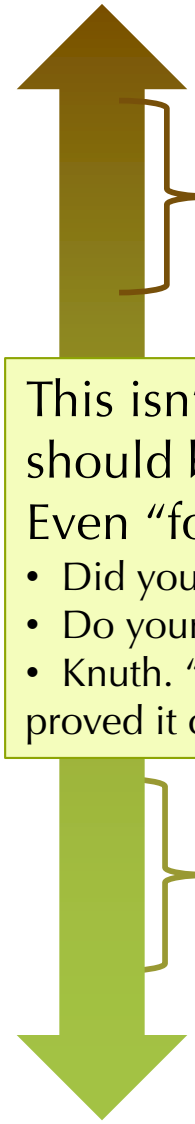
- Yarpgen/YarpgenV2
- ALIVE/ALIVE2 projects
- miscompilation of C, Rust sources [Lee et al. OOPSLA 2018]

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:

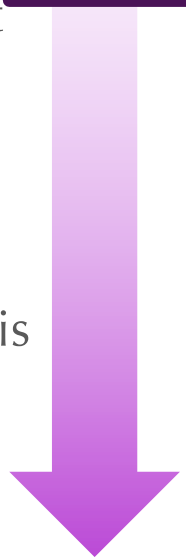
- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth. "Beware of bugs in the above code; I have only proved it correct, not tried it."

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

Goal: Verified Software Correctness

- Social
  - Code reviews
  - Extreme/Pair programmin
- 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?



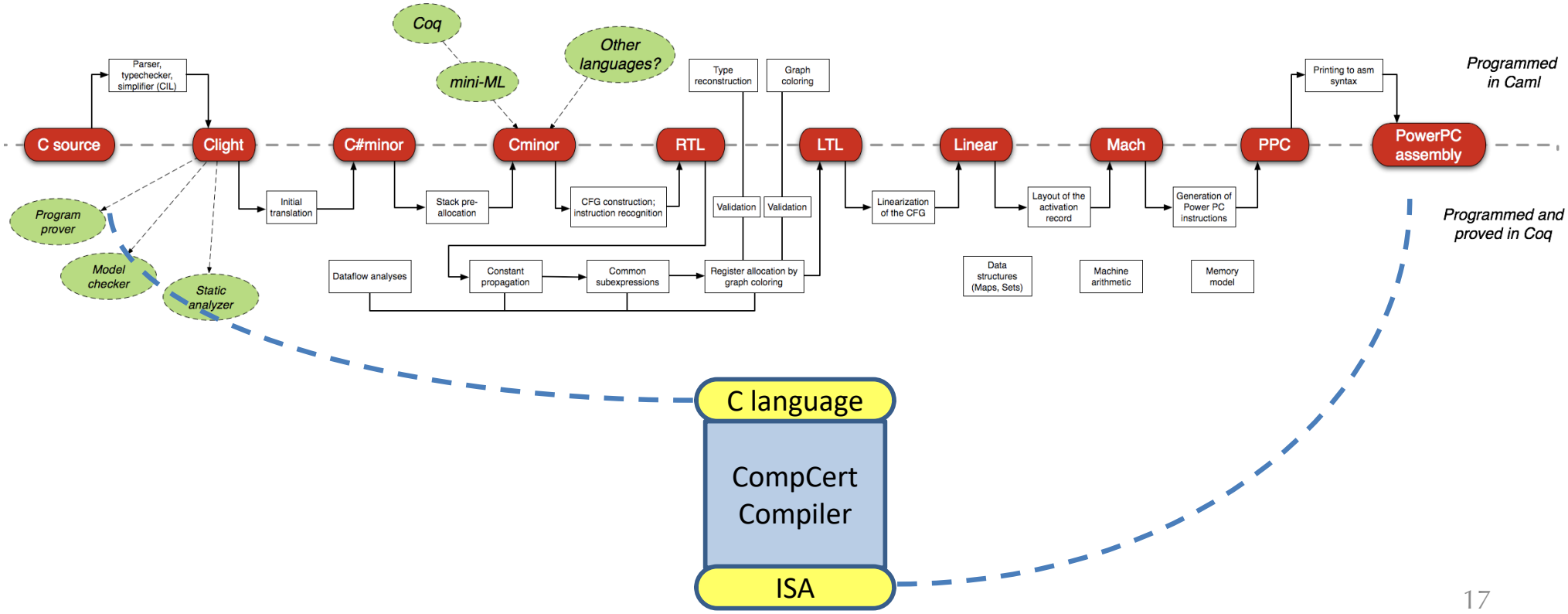
Taking advantage of advances in computer science:

- Moore's law
- improved programming languages & theoretical understanding
- better tools: interactive theorem provers

CompCert – A Verified C Compiler

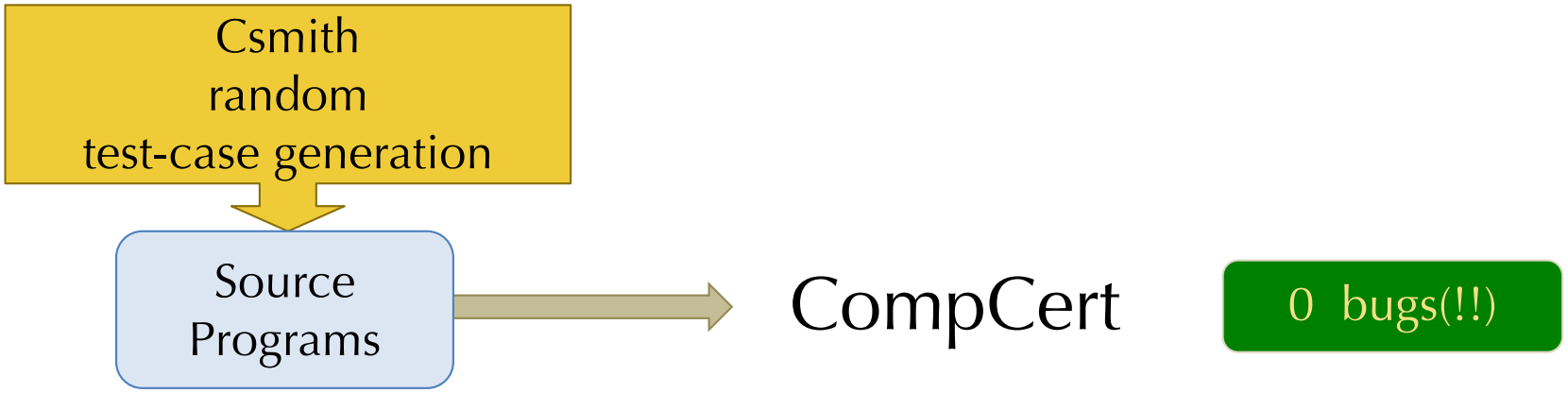
Xavier Leroy
INRIA

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



Csmith on CompCert?

[Yang et al. PLDI 2011]



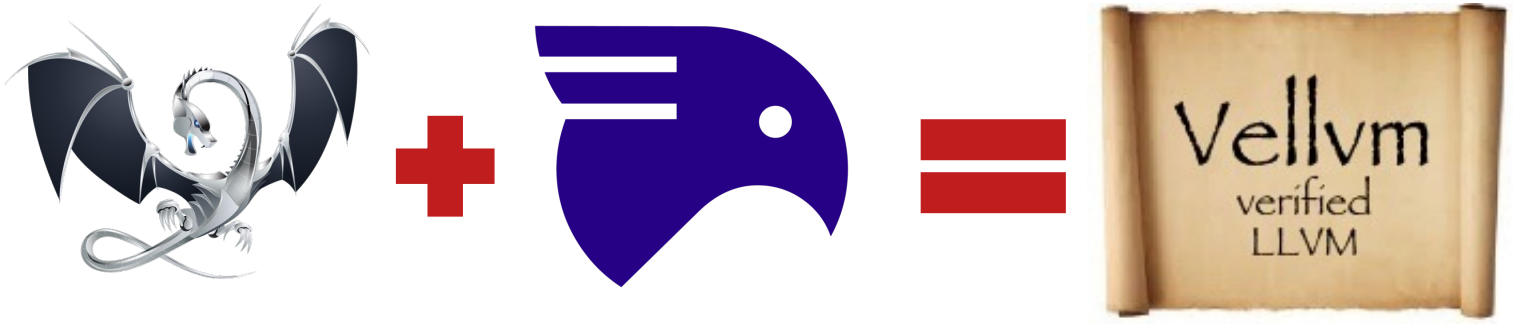
Verification Works!

"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*

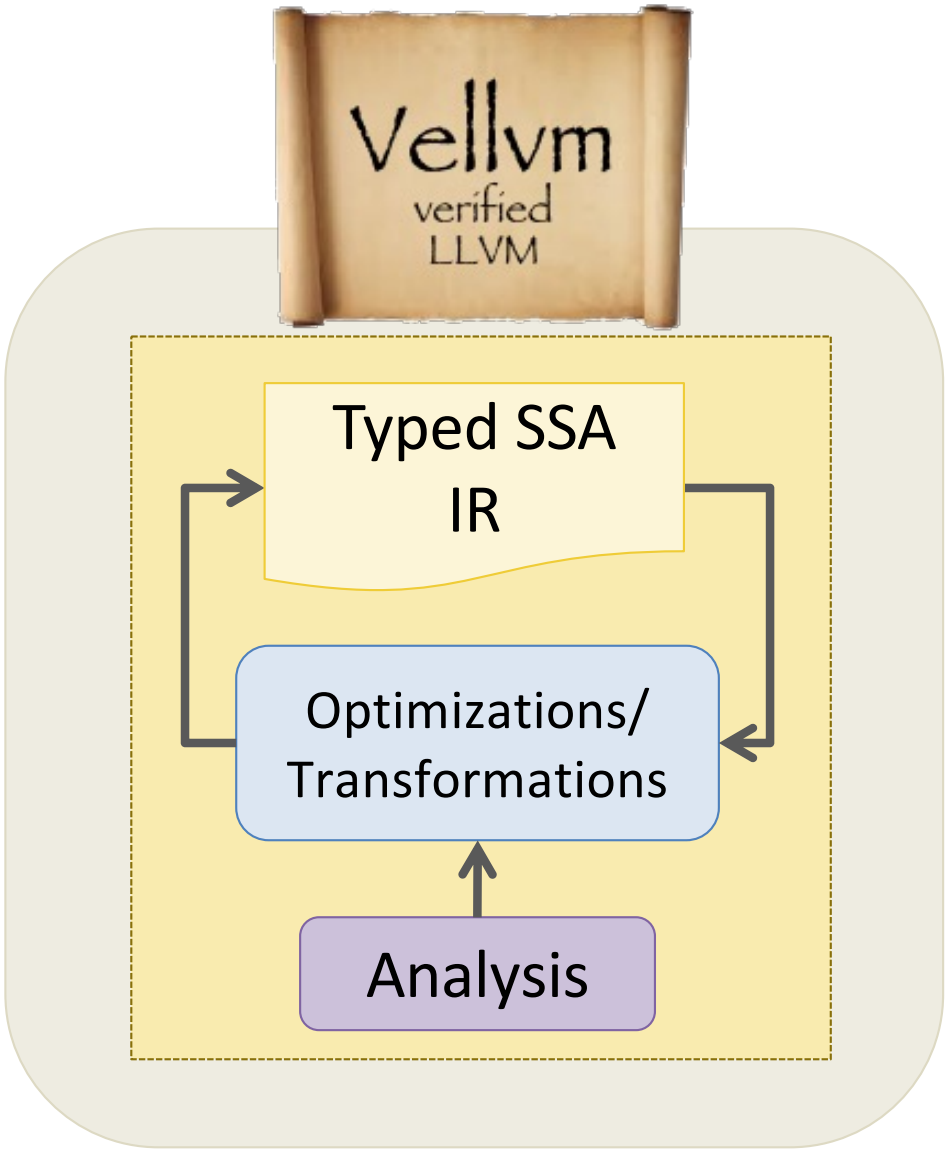
LLVM is hard to trust
(especially for critical code)

What can we do about it?



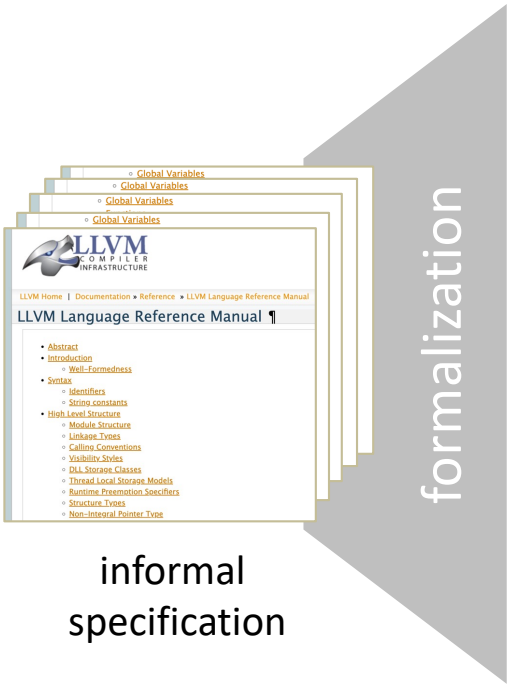
The Vellvm Project

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013, Zackowski, et al. ICFP2021, Beck et al. 2024]



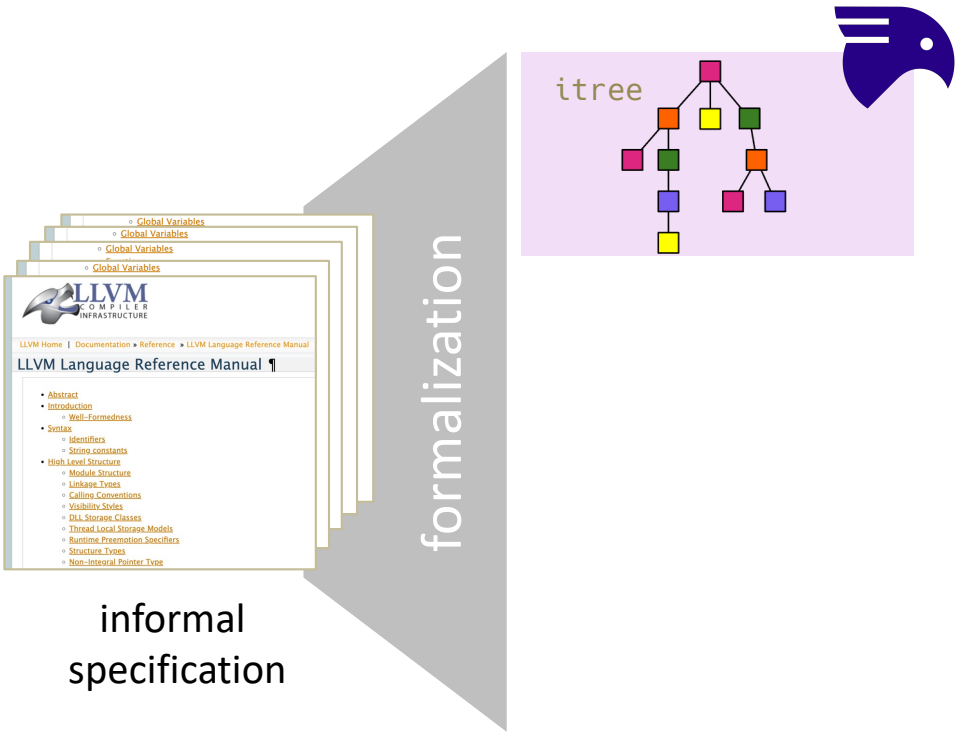
- Formal semantics
- Facilities for creating simulation proofs
- Implemented in Rocq
- Extract passes for use with LLVM compiler
- Example: verified memory safety instrumentation

Structure of the Formalism: Informal Spec



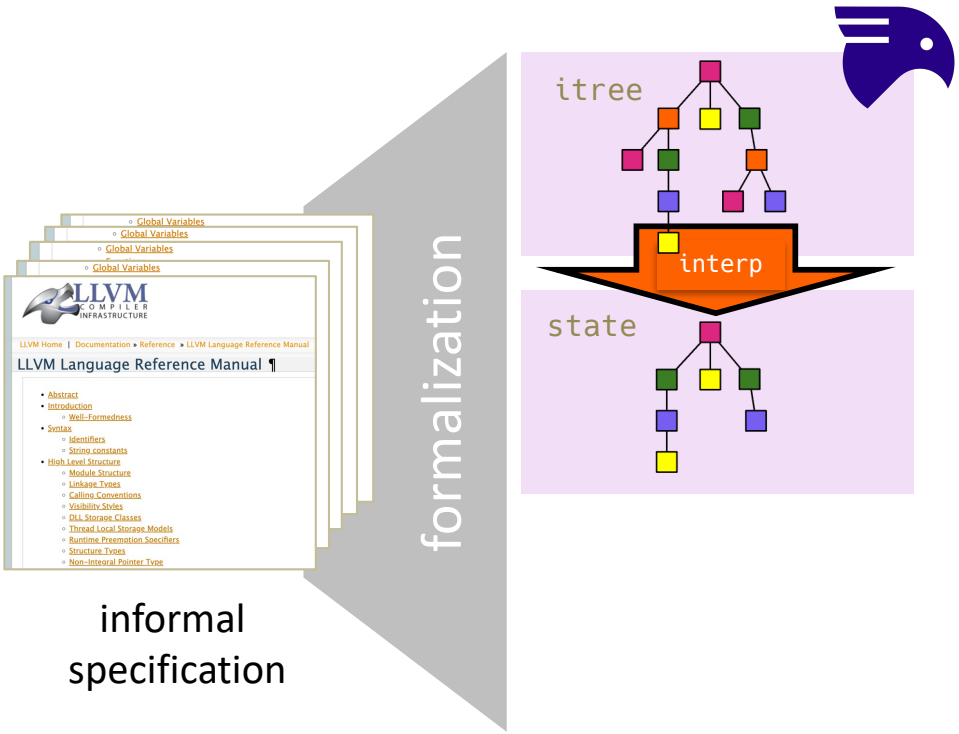
informal specification

Structure of the Formalism: Control Flow



- Abstract Syntax
- Interaction Trees
  - "free monad" (nearly syntax)
  - infinitary behaviors

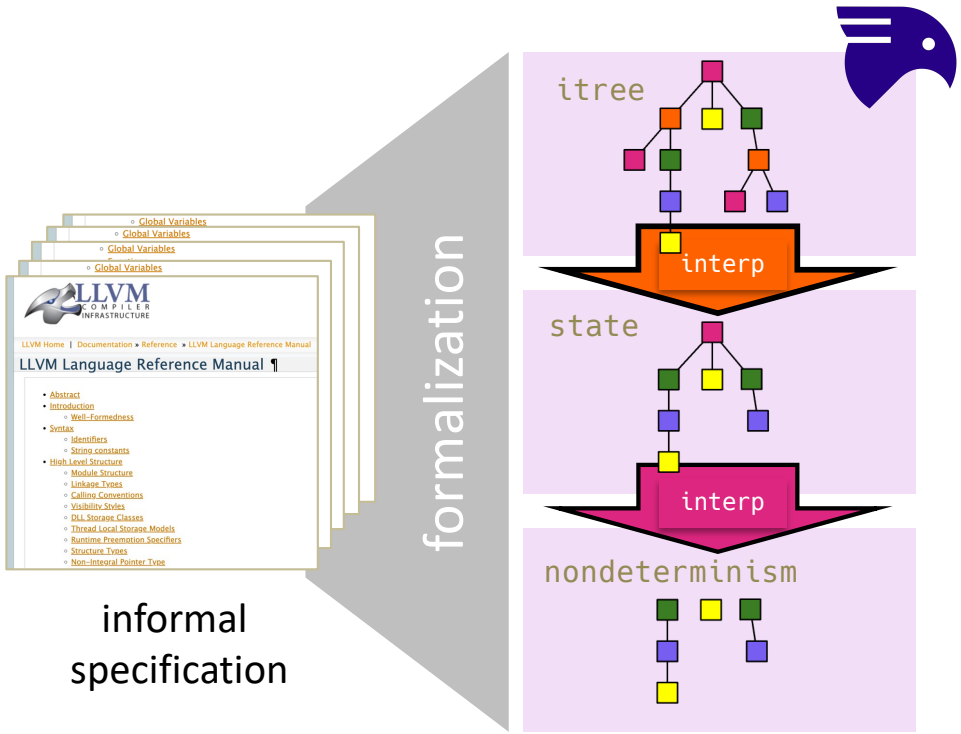
Structure of the Formalism: *Monadic* Interpreters



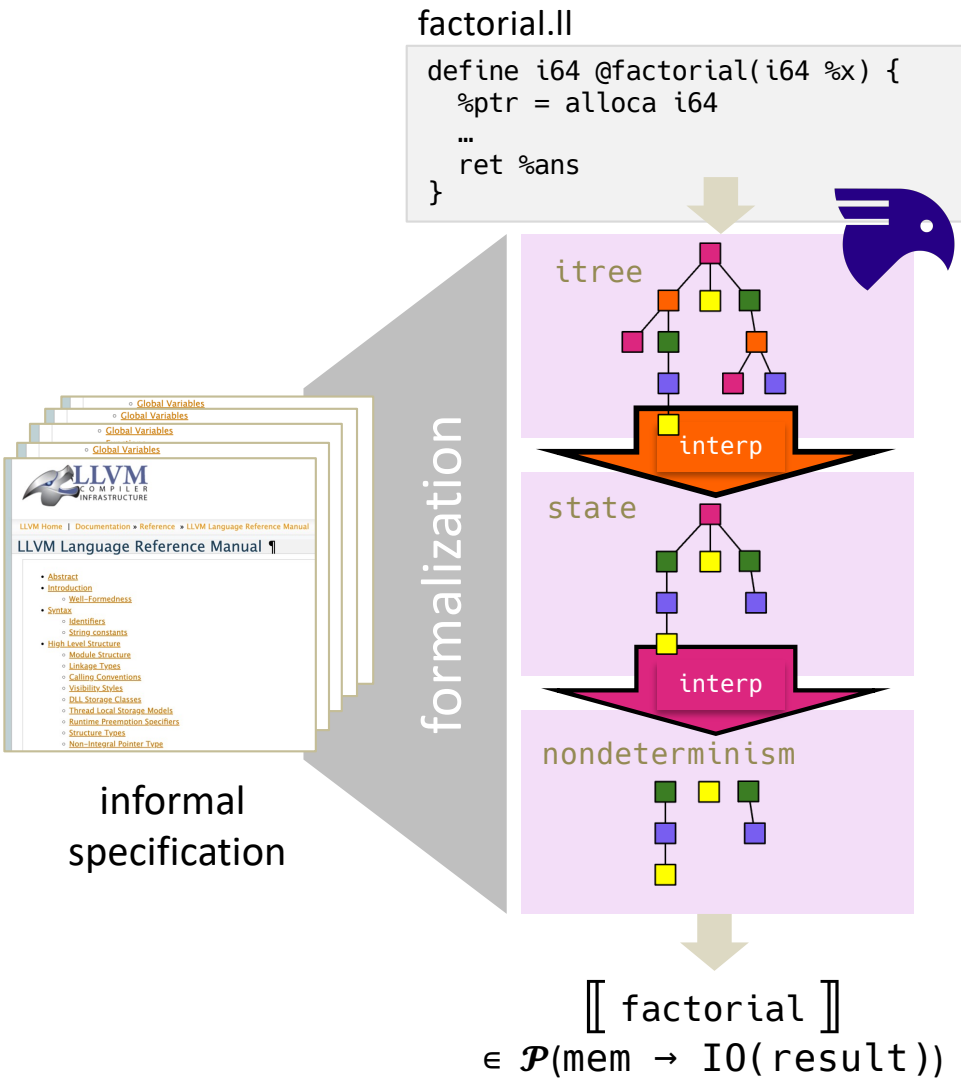
informal specification

- Abstract Syntax
- Interaction Trees
  - "free monad" (nearly syntax)
  - infinitary behaviors
- Modular Event Handlers
  - Interpret the events
  - e.g., to introduce state

Structure of the Formalism: Nondeterminism



- Abstract Syntax
- Interaction Trees
  - "free monad" (nearly syntax)
  - infinitary behaviors
- Modular Event Handlers
  - Interpret the events
  - e.g., to introduce state
- Nondeterminism
  - Interpret into Rocq's `Prop`



Net Result

1. End-to-end specification
   - defines sets of allowed behaviors

2. Prove refinements *between* layers
   - equivalence earlier implies equivalence later

Status of Vellvm

- Large subset of *sequential* LLVM IR
  - Most C programs, modulo libraries
  - A few "intrinsics" + support for adding more
- Rich memory model [ICFP24]
- Verified interpreter with some nice features
  - `printf`
  - Command line argument support for LLVM programs
- GenLLVM ⇒ randomized testing

Upshot: can interpret many compiled C programs, depending on library usage.

Relatively easy to add new intrinsics and library functions.

Writing Interpreters in Rocq

Gallina (Rocq's Language)

- rich, *dependent type* system
- *pure*, *total* functional language
  (all loops must provably terminate)

How do we write the interpretation function?

```
Inductive instr := Load | Store | Add | …
Inductive terminator := Ret | Cbr | Br
Record block :=
   {
    blk_id : block_id;
    blk_code : list (id * instr);
    blk_term : terminator;
   }
```

Datatypes for Abstract Syntax

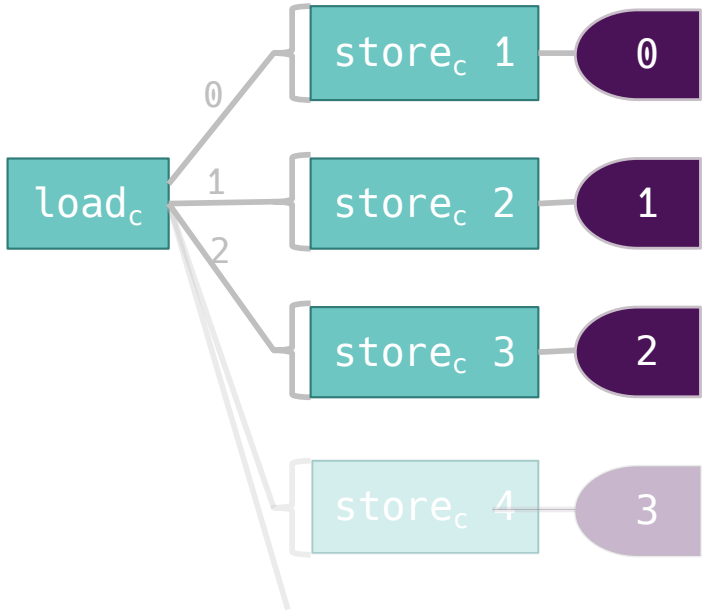
Interaction Trees

[POPL 2020 – Xia et al. Distinguished paper]

Idea: represent (potentially) infinite behaviors as a *data structure*.

```
%x = load i64* %c
%a = add %x 1
store %c, %a
return %x
```

LLVM IR program



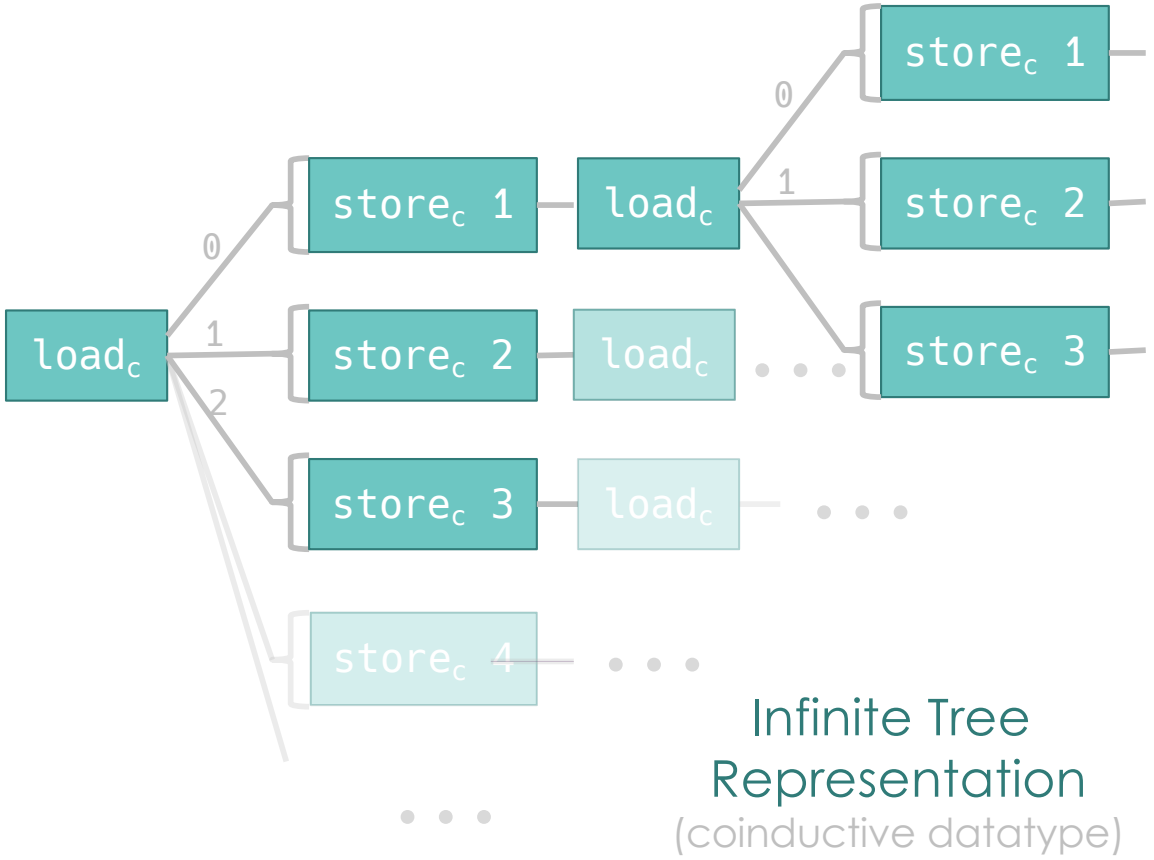
Interaction Tree Representation

Interaction Trees

[POPL 2020 – Xia et al. Distinguished paper]

```
%loop:
  %x = load i64* %c
  %a = add %x 1
  store %c, %a
  br label %loop
```

looping LLVM IR program



Infinite Tree Representation
(coinductive datatype)

Good Qualities of Interaction Trees

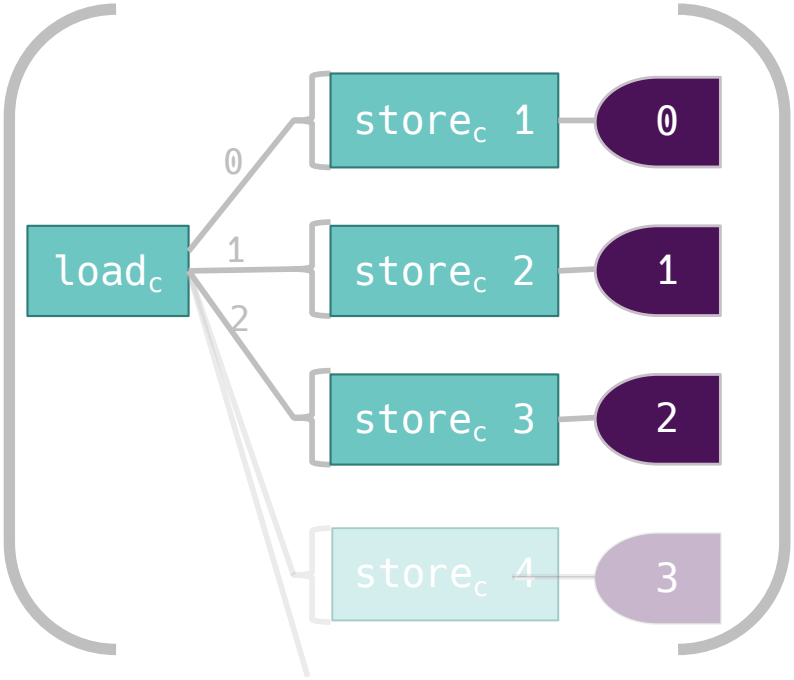
- `ITree E` is a **monad**
  - models sequential computation
  - **`bind`** is grafts on subtrees
- **Extractable** from Rocq into OCaml/Haskell
  ⇒ we can run the computations
- **Behavioral Equivalences**
  - *strong bisimulation*
  - *weak bisimulation*
    (insert finite # taus before any event)
  - rich equational theory

Quite intricate coinductive proofs needed here…
… but, they're encapsulated in the library. [CPP 2020]

Interpreters over ITrees

```
interp_state
```

Process the events of the tree…



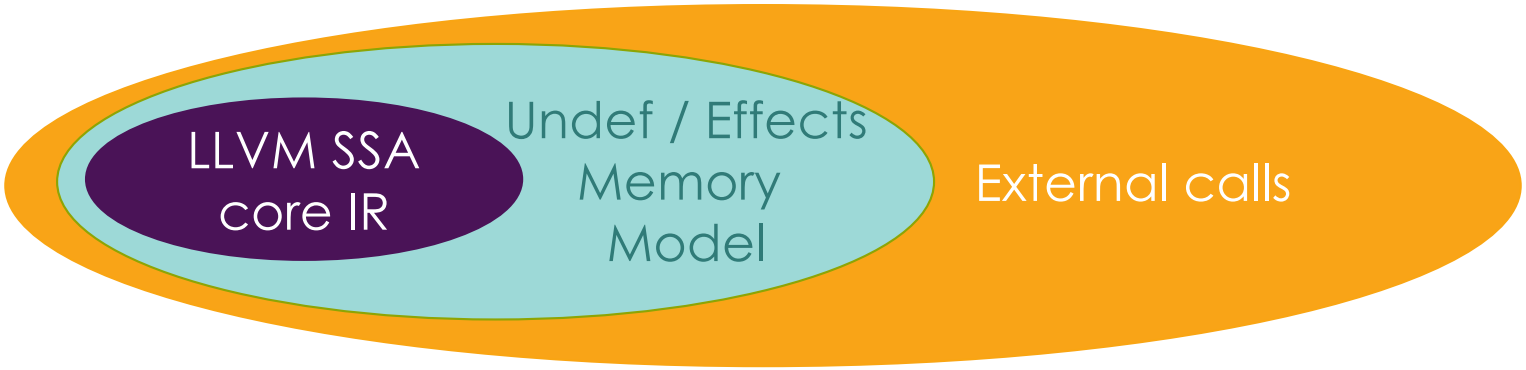
Interaction Tree Representation

```
= fun c ⇒ ret(c+1,x)
```

… to crunch them down to a simple function of the initial state.

Defining Modular Semantics

- Core: LLVM control-flow-graph interpreter
- `undef` values / computation
- Memory Model [CAV 15,PLDI 15, ICFP24]
  - interprets loads/stores
  - nondeterminism for allocation
  - support for casts

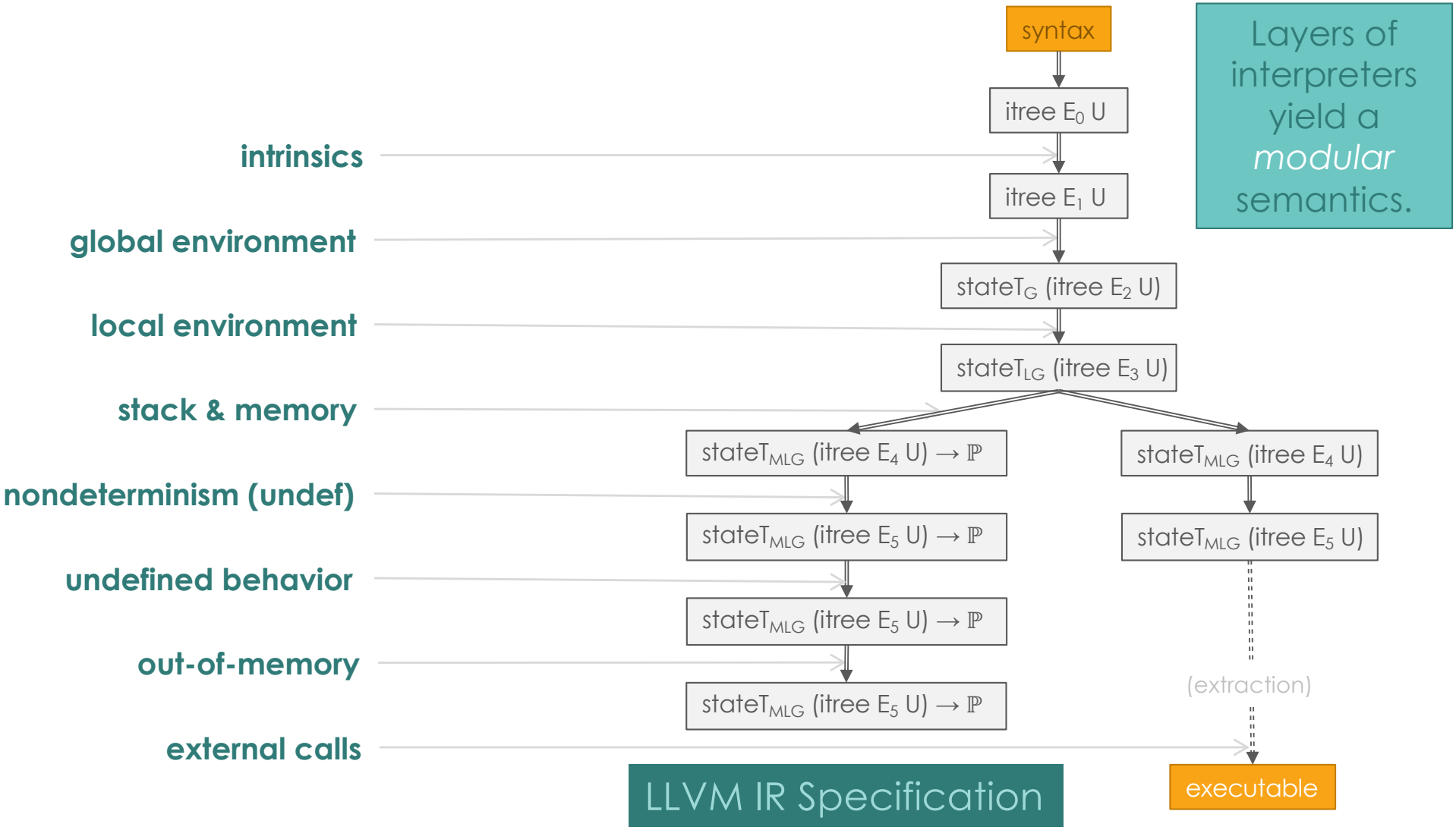


LLMV Memory Model Events
(simplified)

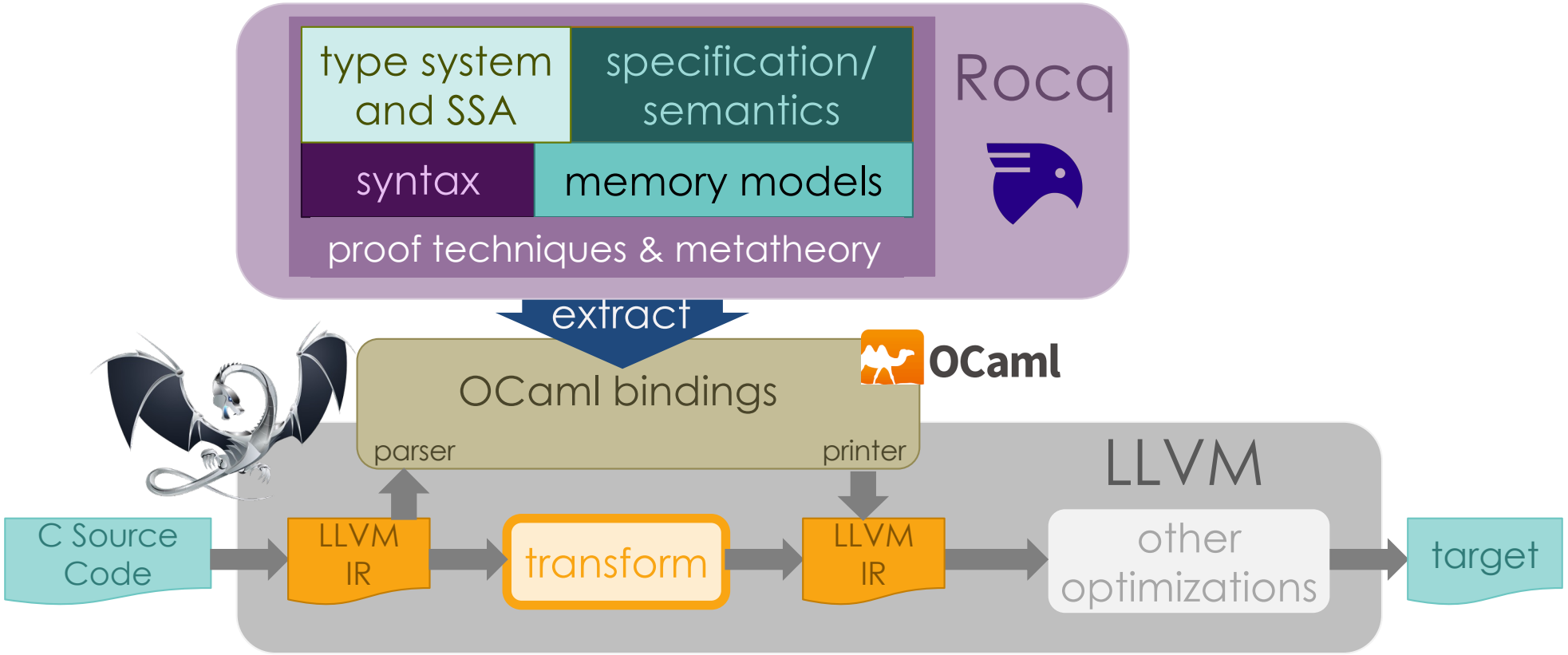
```
(* Memory event types for the LLVM IR *)
Inductive MemE :=
| Alloca  : type                  → MemE uvalue
| Load    : type   → uvalue       → MemE uvalue
| Store   : uvalue → uvalue       → MemE unit
| MemPush :                       MemE unit
| MemPop  :                       MemE unit.
```

"uvalue" is a nondeterministic computation that involves undef

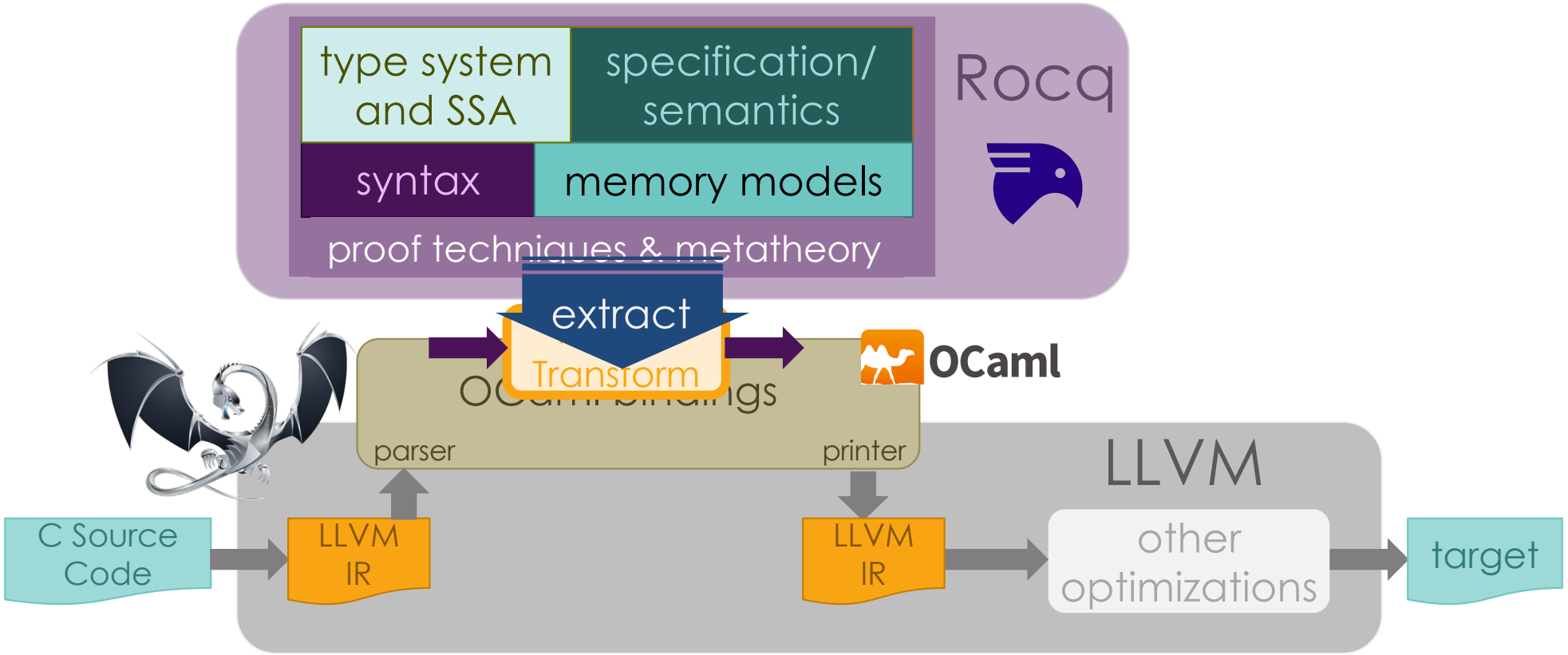
LLVM IR Effects



Vellvm Framework



Vellvm Framework



So What?

- **Finding bugs**
  - thinking hard about corner cases
  - identify inconsistent assumptions of the LLVM compiler
- **Automated Tests**
  - e.g. integrate with CSmith
- **Formally validate program transformations**
  - is a particular optimization correct?
  - improved confidence
- **Verified compilers**
  - to obtain high-confidence software

Ambiguities in LLVM IR

"Allocating zero bytes is legal, but the **returned pointer may not be unique**"
-- LangRef

"The allocated memory is uninitialized, and **loading from uninitialized memory produces an undefined value**"
-- LangRef

```
%p = alloca [0 x i32]
%x = load [0 x i32], [0 x i32]* %p
```

Is %x undef?

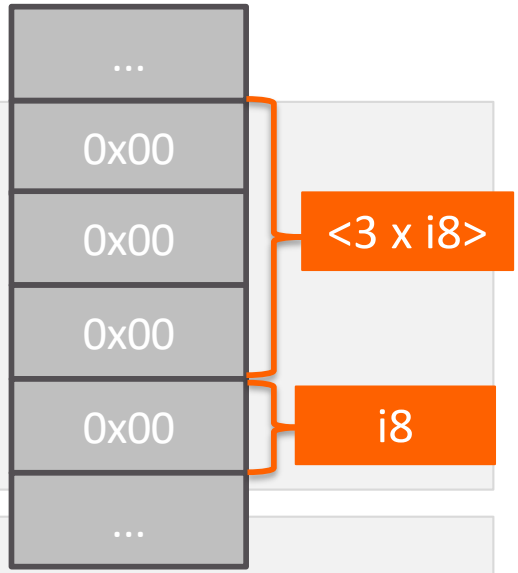
We Found Bugs in Clang!

```
define i8 @main() {
    %v0 = alloca i32
    store i32 0, i32* %v0, align 1
    %v1 = ptrtoint i32* %v0 to i64
    %v2 = inttoptr i64 %v1 to <{T, i8}>*
    %v3 = load <{T, i8}>, <{T, i8}>* %v2, align 1
    %v4 = extractvalue <{T, i8}> %v3, 1
    ret i8 %v4
}
```

clang -S

T = <3 x i8>

```
main:
    addi    sp, sp, -16
    mv      a0, zero
    sw      a0, 12(sp)
    lb      a0, 16(sp)
    addi    sp, sp, 16
    ret
```



- Off-by-one error in extractvalue with packed struct with vector

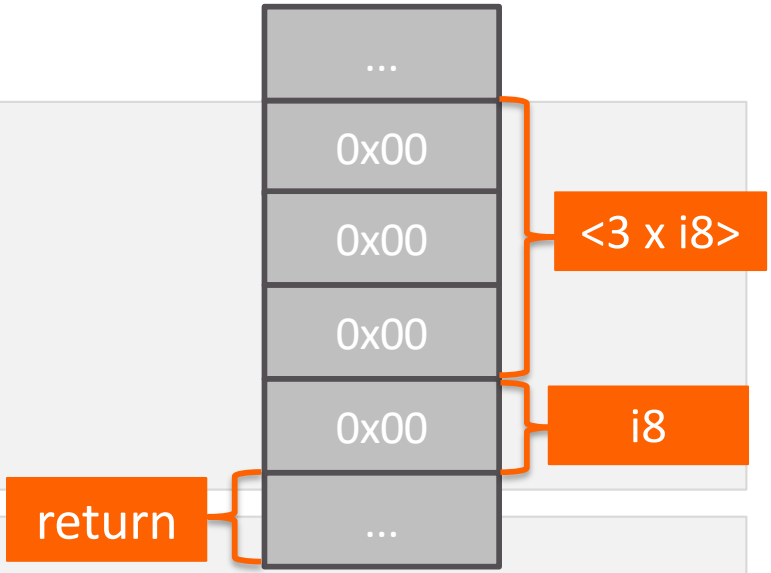
We Found Bugs in Clang!

```
define i8 @main() {
    %v0 = alloca i32
    store i32 0, i32* %v0, align 1
    %v1 = ptrtoint i32* %v0 to i64
    %v2 = inttoptr i64 %v1 to <{T, i8}>*
    %v3 = load <{T, i8}>, <{T, i8}>* %v2, align 1
    %v4 = extractvalue <{T, i8}> %v3, 1
    ret i8 %v4
}
```

clang -S

T = <3 x i8>

```
main:
    addi    sp, sp, -16
    mv      a0, zero
    sw      a0, 12(sp)
    lb      a0, 16(sp)
    addi    sp, sp, 16
    ret
```



- Off-by-one error in extractvalue with packed struct with vector

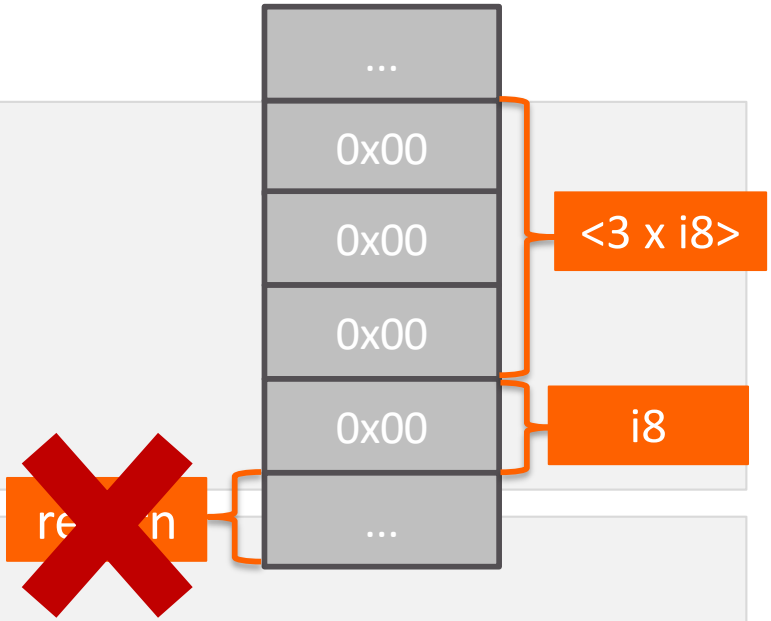
We Found Bugs in Clang!

```
define i8 @main() {
    %v0 = alloca i32
    store i32 0, i32* %v0, align 1
    %v1 = ptrtoint i32* %v0 to i64
    %v2 = inttoptr i64 %v1 to <{T, i8}>*
    %v3 = load <{T, i8}>, <{T, i8}>* %v2, align 1
    %v4 = extractvalue <{T, i8}> %v3, 1
    ret i8 %v4
}
```

clang -S

T = <3 x i8>

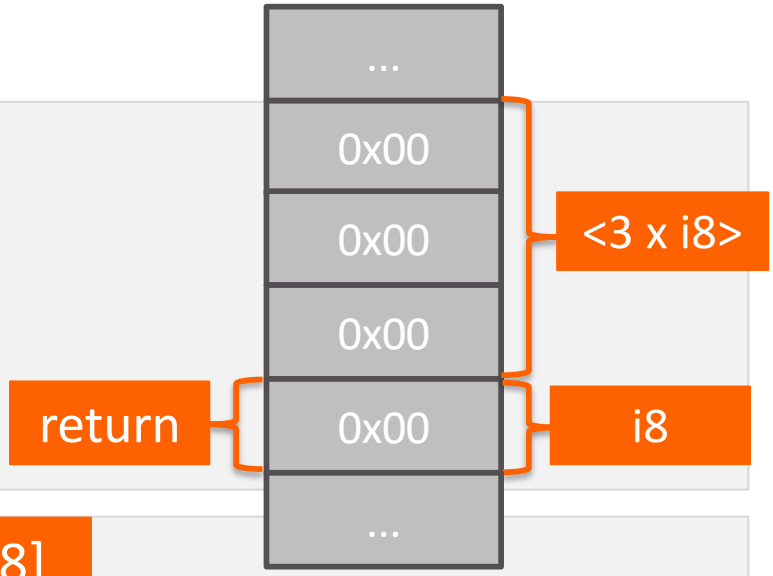
```
main:
    addi    sp, sp, -16
    mv      a0, zero
    sw      a0, 12(sp)
    lb      a0, 16(sp)
    addi    sp, sp, 16
    ret
```



- Off-by-one error in extractvalue with packed struct with vector

We Found Bugs in Clang!

```
define i8 @main() {
    %v0 = alloca i32
    store i32 0, i32* %v0, align 1
    %v1 = ptrtoint i32* %v0 to i64
    %v2 = inttoptr i64 %v1 to <{T, i8}>*
    %v3 = load <{T, i8}>, <{T, i8}>* %v2, align 1
    %v4 = extractvalue <{T, i8}> %v3, 1
    ret i8 %v4
}
```



clang -S

T = <3 x i8>

```
main:
    addi    sp, sp, -16
    mv      a0, zero
    sw      a0, 12(sp)
    lb      a0, 16(sp)
    addi    sp, sp, 16
    ret
```

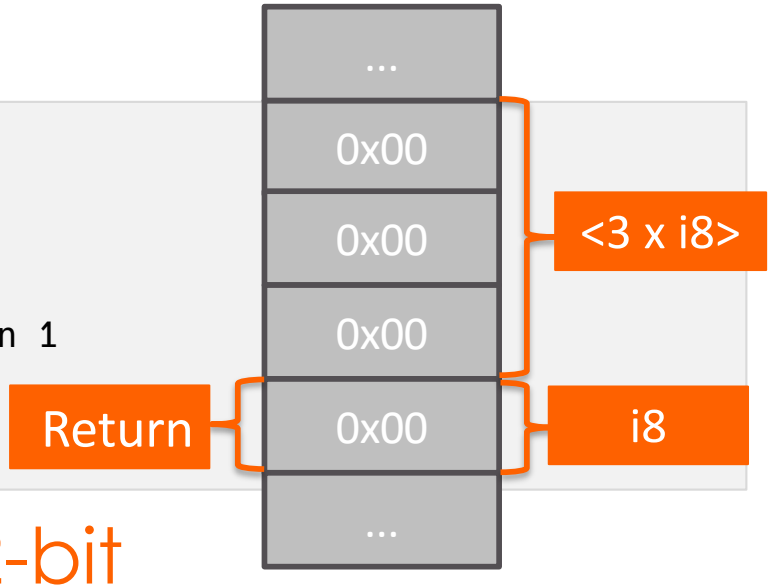
T = [3 x i8]

```
main:
    addi    sp, sp, -16
    mv      a0, zero
    sw      a0, 12(sp)
    lb      a0, 15(sp)
    addi    sp, sp, 16
    ret
```

- Off-by-one error in extractvalue with packed struct with vector

We Found Bugs in Clang!

```
define i8 @main() {
    %v0 = alloca i32
    store i32 0, i32* %v0, align 1
    %v1 = ptrtoint i32* %v0 to i64
    %v2 = inttoptr i64 %v1 to <{<3 x i8>, i8}>*
    %v3 = load <{<3 x i8>, i8}>, <{<3 x i8>, i8}>* %v2, align 1
    %v4 = extractvalue <{<3 x i8>, i8}> %v3, 1
    ret i8 %v4
}
```



- "zero-initialize" an i32 into a 32-bit packed-struct
- Read the last byte from it
- Semantically, 0 should be returned!

But:

```
clang: 46

Vellvm: 0
```

Proving Optimizations:

- **Refinement of uvalues**

  ⟦i8 undef⟧ = {0,…,255}
  ⟦i8 1⟧ = {1}
  so: ⟦i8 undef⟧ ⊇ ⟦i8 1⟧

We can soundly "implement" the specification value `undef` with 1.

- **Behavioral refinement of computations**

$$⟦P⟧_m \supseteq ⟦Q⟧_m$$

Q is a valid optimization of P.

Proofs *are* programs!

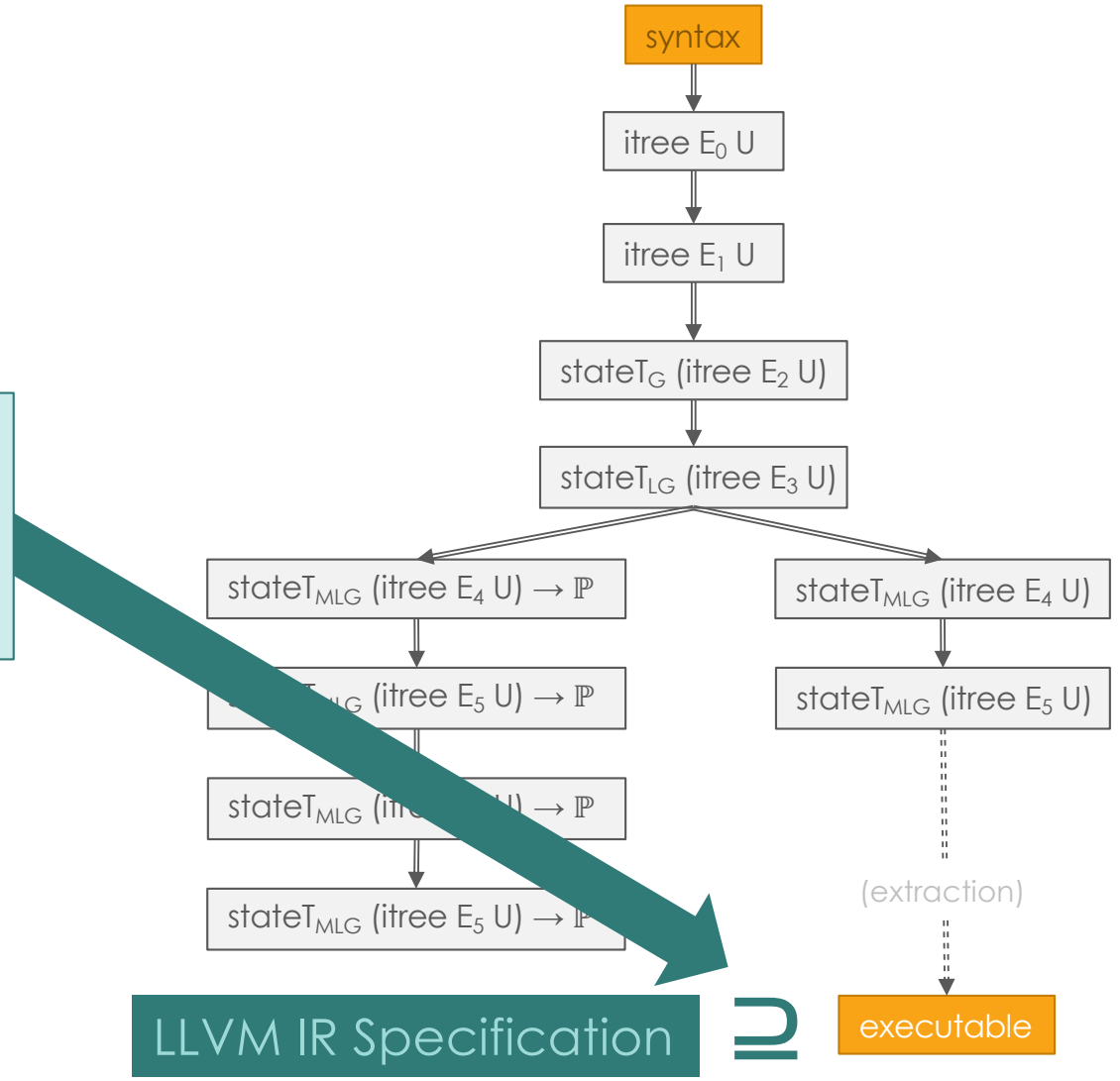
```
Theorem mul_add_correct :
  ∀ x, x ≠ undef → ⟦mul i64 2, x⟧ ⊇ ⟦add i64 x, x⟧.
Proof.
  …
Qed.
```

Machine checked proof! We can't miss cases or use an invalid step of reasoning.

We can soundly optimize multiplication to addition, assuming x is defined.

Correctness Theorem

Theorem: *The executable reference interpreter is a valid implementation of the specification.*



Example Optimizations

- **Standard "computational" optimizations**
  - e.g., ⟦`mul i64 %x, undef`⟧ ⊇ {`0`}
- **Control-flow graph optimizations**
  - e.g. "block fusion" = merge two sequential blocks
- **Dead-alloca / dead ptrtoint cast elimination**

```
define void @ptoi_code() {
  %ptr = alloca i64
  %i = ptrtoint i64* to i64
  ret void
}
```

⊇

```
define void @ret_code() {
  ret void
}
```

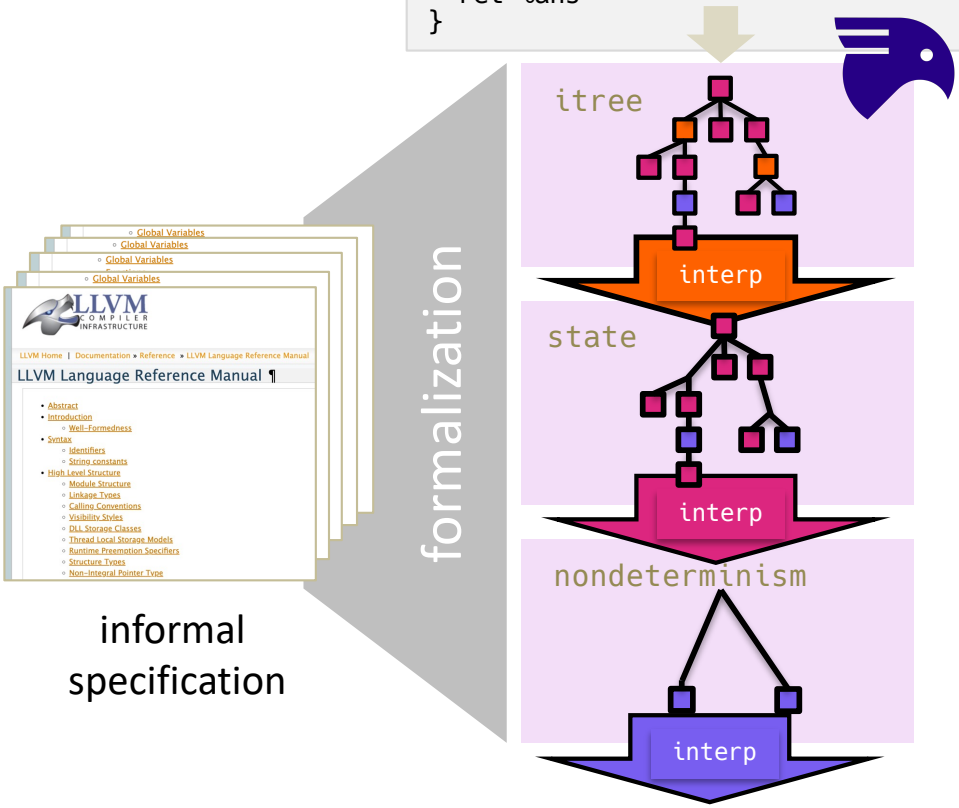
Surprisingly difficult to prove! [Beck et al. 2024]
- infinite memory ⇒ ptrtoint removal unsound!
- finite memory ⇒ alloca removal unsound!

Validity of Vellvm

- We have a large formal semantics for LLVM IR
  - Formal interpreter using interaction trees and a two-phase infinite/finite memory model.
- But does it adhere to the LLVM Language Reference? 🧐
  - We want Vellvm semantics faithfully model the informal specification.
- Testing!
  - Unit Testing (manual + Alive2)
  - Differential Testing (like CSmith and YARPGen)

factorial.ll

```
define i64 @factorial(i64 %x) {
  %ptr = alloca i64
  …
  ret %ans
}
```



informal specification

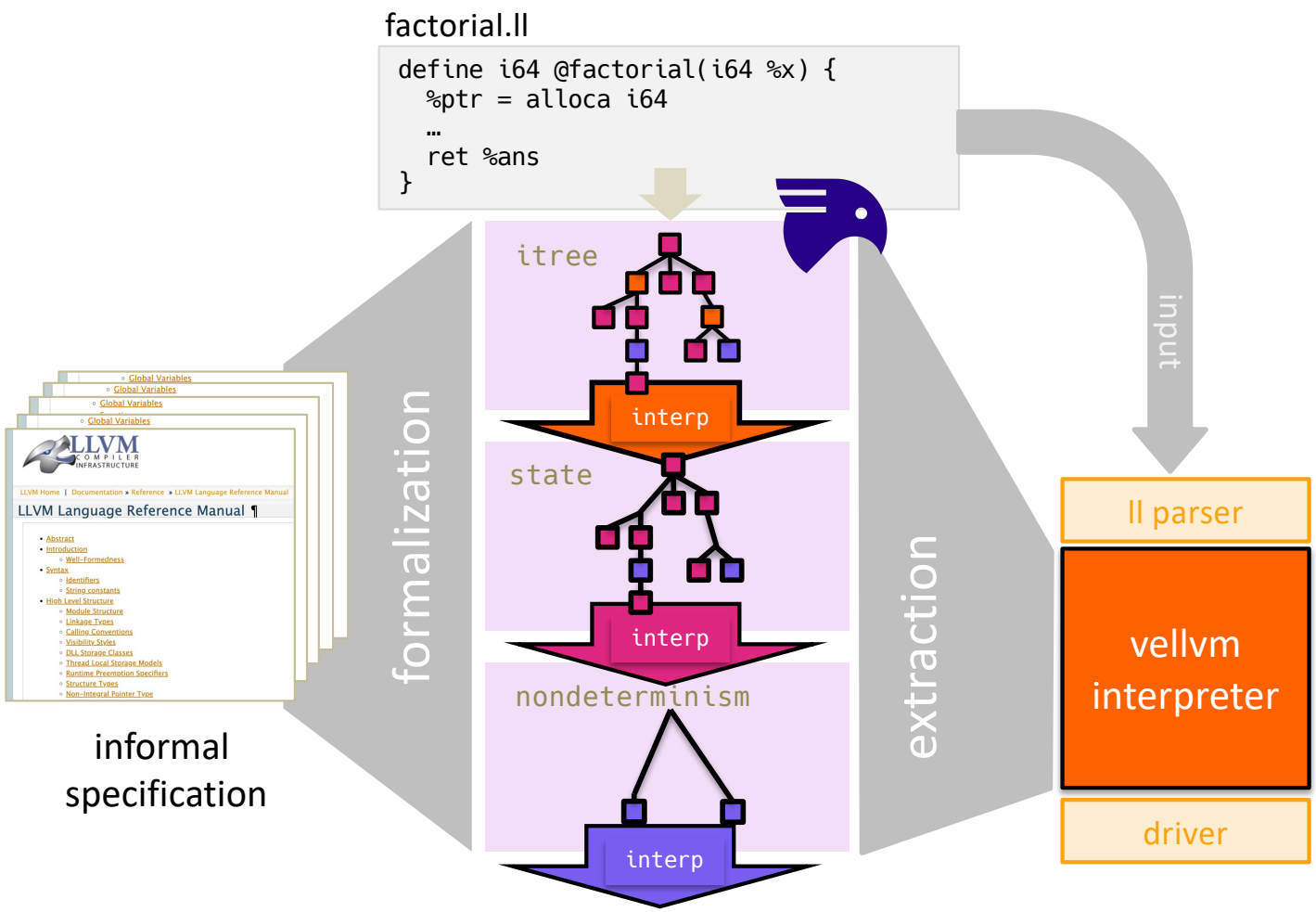
formalization

$[\![\texttt{factorial}]\!]$

$\in \mathcal{P}(\texttt{mem} \to \texttt{IO(result)})$

factorial.ll

```
define i64 @factorial(i64 %x) {
  %ptr = alloca i64
  …
  ret %ans
}
```



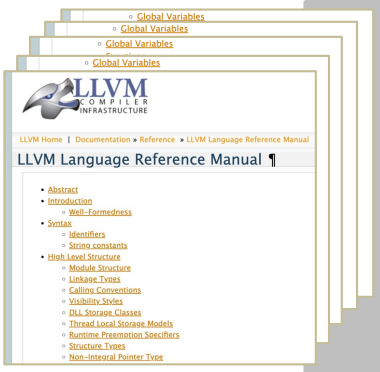
informal specification

$[\![\texttt{factorial}]\!] \in \mathcal{P}(\texttt{mem} \to \texttt{IO(result)})$

```
> vellvm -interpret factorial.ll
Program terminated with: i64 120
```

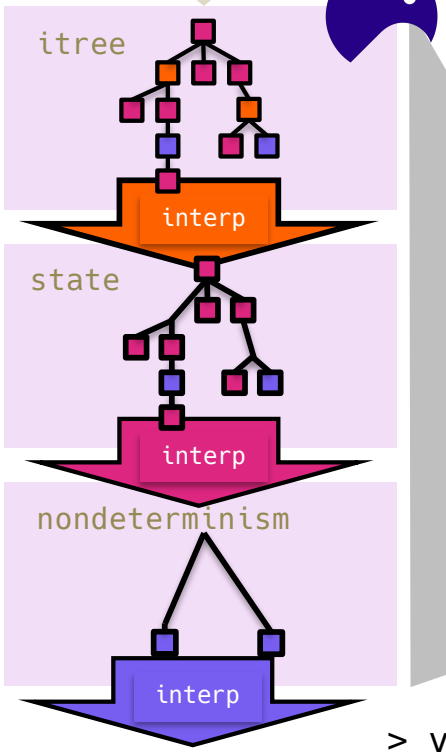
factorial.ll

```
define i64 @factorial(i64 %x) {
  %ptr = alloca i64
  …
  ret %ans
}
```



informal specification

formalization



extraction

input

input

ll parser

vellvm interpreter

driver

Differential Testing

llc

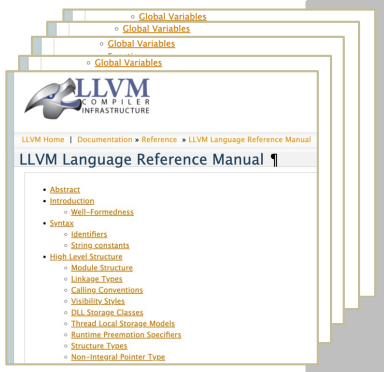
```
> vellvm -interpret factorial.ll
Program terminated with: i64 120
```

```
> clang factorial.ll
> ./a.out
120
```

⟦ factorial ⟧ ∈ $\mathcal{P}$(mem → IO(result))

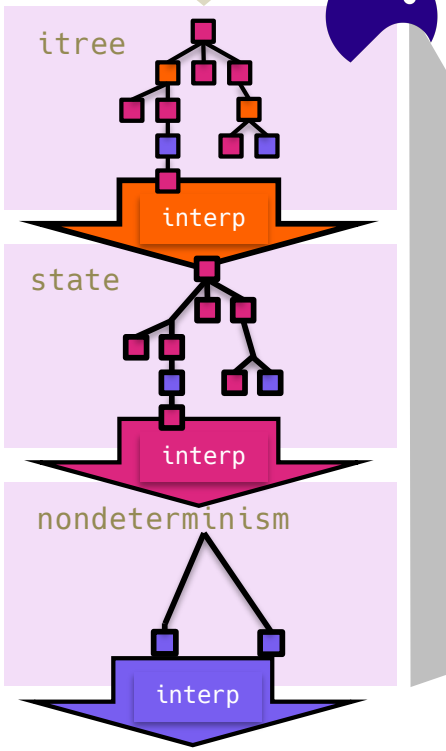
factorial.ll

```
define i64 @factorial(i64 %x) {
  %ptr = alloca i64
  …
  ret %ans
}
```



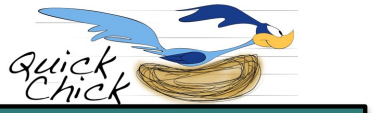
informal specification

formalization



extraction

⟦ factorial ⟧
∈ $\mathcal{P}$(mem → IO(result))



tst.ll

input

input

input

ll parser

vellvm interpreter

driver

differential testing

llc

```
> vellvm -interpret tst.ll
Program terminated with: i8 100
```

```
> clang tst.ll
> ./a.out
80
```

Trickiness in Differential Testing of LLVM IR

Requirements for LLVM Programs

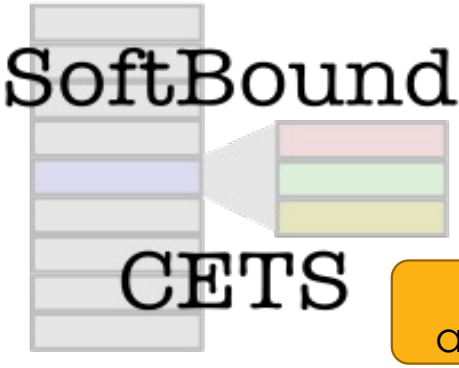
- Deterministic
- Well-formed
  - Type-safe, well-scoped, etc.
- Undefined-behavior free

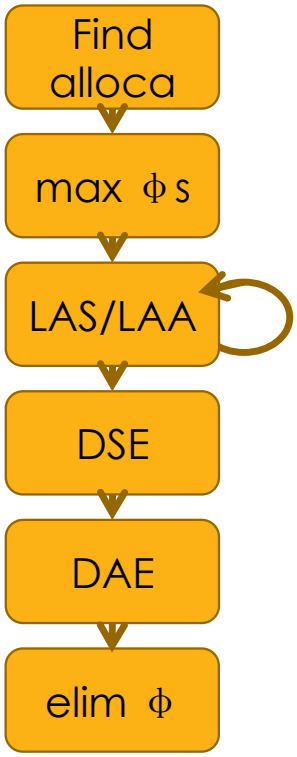
Desirable Properties for Testing

- Exercise large subset of LLVM syntax
- Nontrivial control flows
- Random code patterns

VELLVM

- **Verified SoftBound** [POPL 2012]
  - retrofit memory safety in legacy C code
- **Verified mem2reg** [PLDI 2013]
  - register promotion, defined via "micro-optimizations"
- **Verified Program Analyses** [CPP 2012]
  - Cooper-Harvey-Kennedy Dominator Algorithm [2000]
- **Memory models**
  - ptrtoint casts [PLDI 2015]
  - modular formalization [CAV 2015]
  - infinite/finite reconciliation [ongoing]





Vellvm: Ongoing / Future Work

- Better I/O / C library support
- Factoring out the memory model
- Verified Analyses and Optimizations
- Concurrency
  - "A concurrency model based on monadic interpreters"
    [Chappe et al. CPP25]
- LLVM IR version bump
- More instructions in GenLLVM

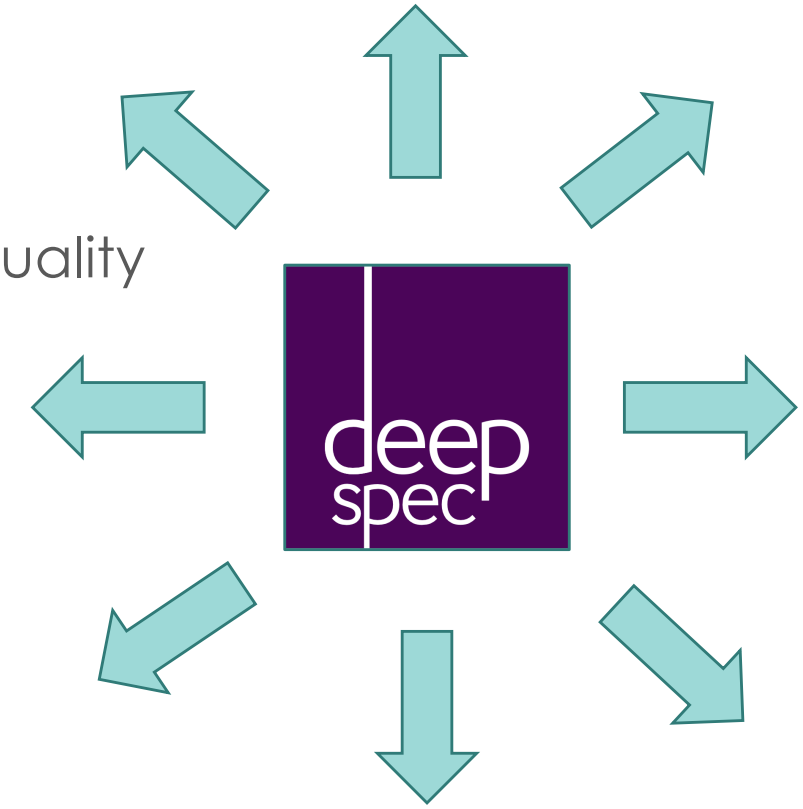
https://github/Vellvm/vellvm

Can Formal Verification Scale?

- **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 systems/packet filters
  Chlipala, MIT
- **CertiKOS** – certified OS kernel
  Shao & Ford, Yale
- **CakeML** – certified ML compiler
- **Core Spec** – Haskell
- **SEL4** – certified secure OS microkernel
- **Kami** – verified RISCV architecture
- **Cryptography** – verified SSL primitives

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, $$$$
- **Formal Specifications**
  - what are the principles?
  - compositionality?



What have we learned?

Where else is it applicable?

What next?

COURSE WRAP-UP

Final Exam

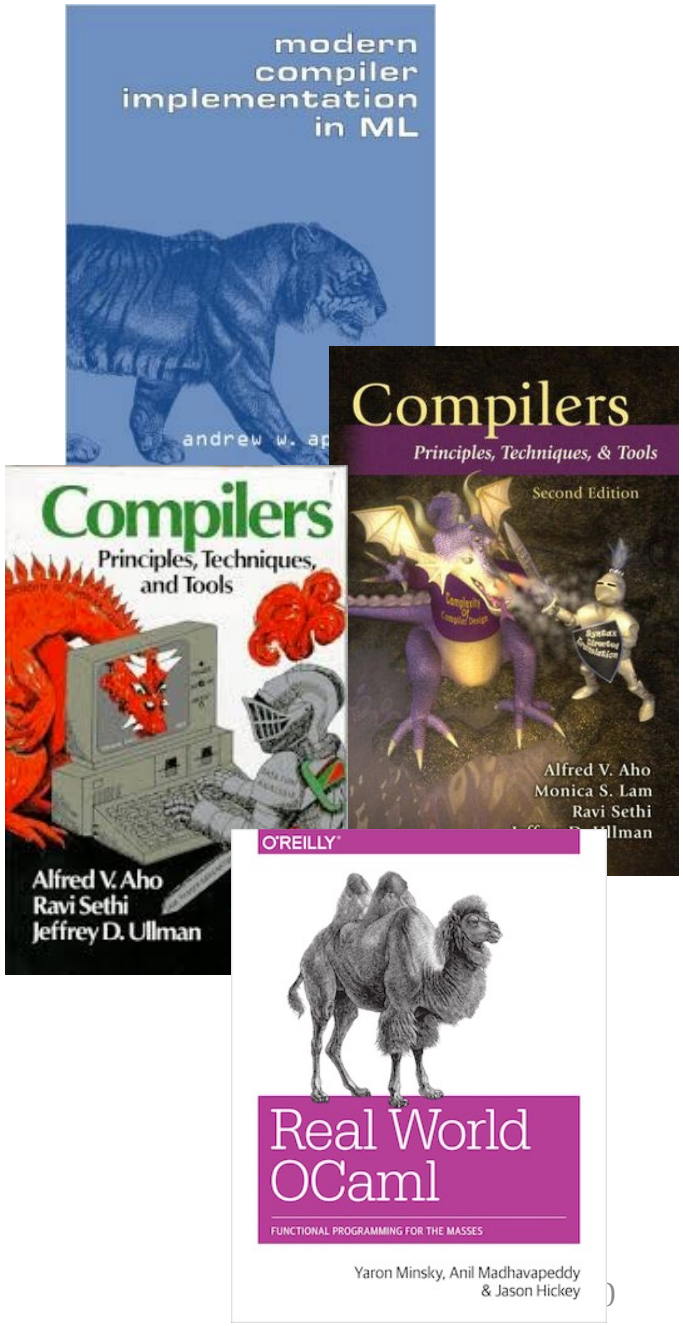
- Will mostly cover material since the midterm
  - Starting from Lecture 14
  - Lambda calculus / closure conversion
  - Scope / Typechecking / Inference Rules
  - Objects, inheritance, types, implementation of dynamic dispatch (de-emphasized, since we didn't cover it thoroughly)
  - Basic optimizations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

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

Why CIS 4521/5521?

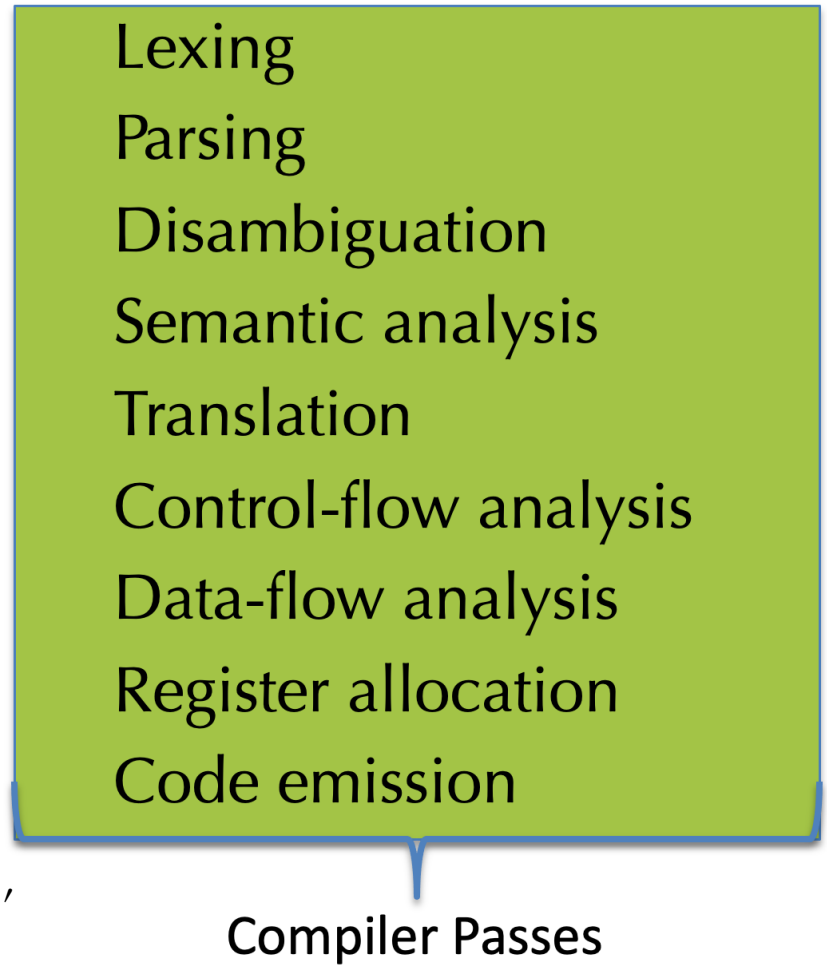
- You will learn:
  - Practical applications of theory
  - Parsing
  - How high-level languages are implemented in machine language
  - (A subset of) Intel x86 architecture
  - A deeper understanding of code
  - A little about programming language semantics
  - Functional programming in OCaml
  - How 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



Related Courses

- CIS 5000: Software Foundations
  - Theoretical course about functional programming, proving program properties, type systems, lambda calculus. Uses the theorem prover Rocq.
- CIS 5010: Computer Architecture
  - Prof. Devietti
  - 4710++: pipelining, caches, VM, superscalar, multicore,…
- CIS 5470: Software Analysis
  - Prof. Naik
  - LLVM IR + program analysis
- CIS 5520: Advanced Programming
  - Prof. Weirich
  - Advanced functional programming in Haskell, including generic programming, metaprogramming, embedded languages, cool tricks with fancy type systems
- CIS 6700: Special topics in programming languages
- CIS 7000: Programming Languages: Semantics and Types

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
  - MLIR – based on LLVM IR, but more general-purpose
  - 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
  - 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: **Gary Chen**, **Noé De Santo**, and **Alex Shypula**
- To *you* for taking the class!
- How can I improve the course?
  - Let me know in course evaluations!