Vellvm:
Formalizing the Informal

Calvin Beck, **Hanxi (Gary) Chen**, Steve Zdancewic
University of Pennsylvania

CoqPL Workshop 2025

Thanks to these Collaborators!

Calvin Beck
Hanxi (Gary) Chen
Dmitri Garbuzov
Olek Gierczak
Paul He
Gil Hur
William Mansky
Milo M.K. Martin
Noé De Silva
Roger Burtonpatel
Santosh Nagarakatte
Gregory Malecha
Benjamin Pierce
Christine Rizkallah
Lucas Silver
Li-yao Xia
Irene Yoon
Yannick Zakowski
Jianzhou Zhao



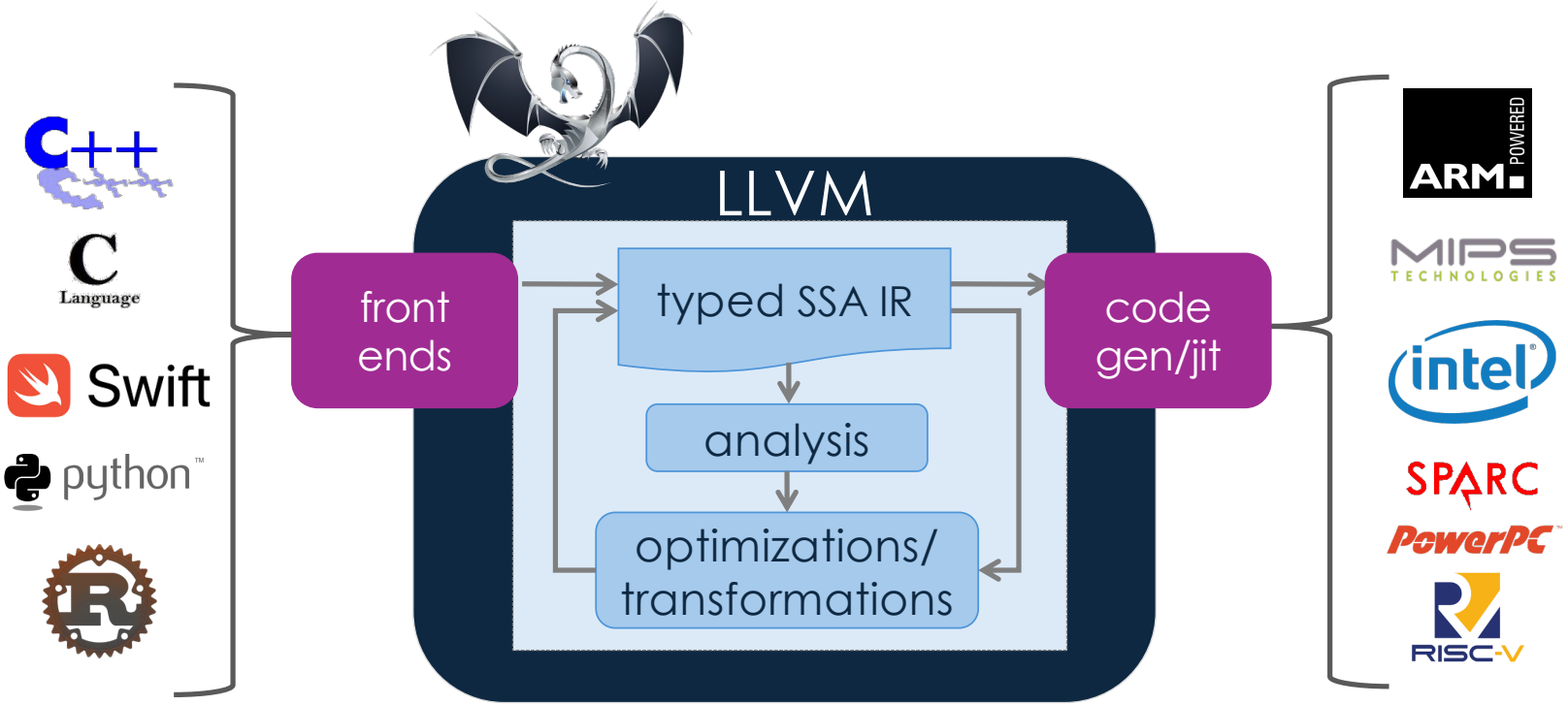


Outline

- **LLVM IR**
  - **pain points: undef ptrtoint/ lack of intptr type, varargs, platform-specific**
    undef + memory
- Vellvm
  - Sketch of Formalization
- Rocq Experience
- Differential Testing
  - GenLLVM structure
  - results
- Future

LLVM Compiler Infrastructure

[Lattner,2002]



Translating C to LLVM IR

```
factorial(int n) {
  int acc = 1;
  while(n > 0) {
    acc = acc * n;
    n--;
  }
  return acc;
}
```

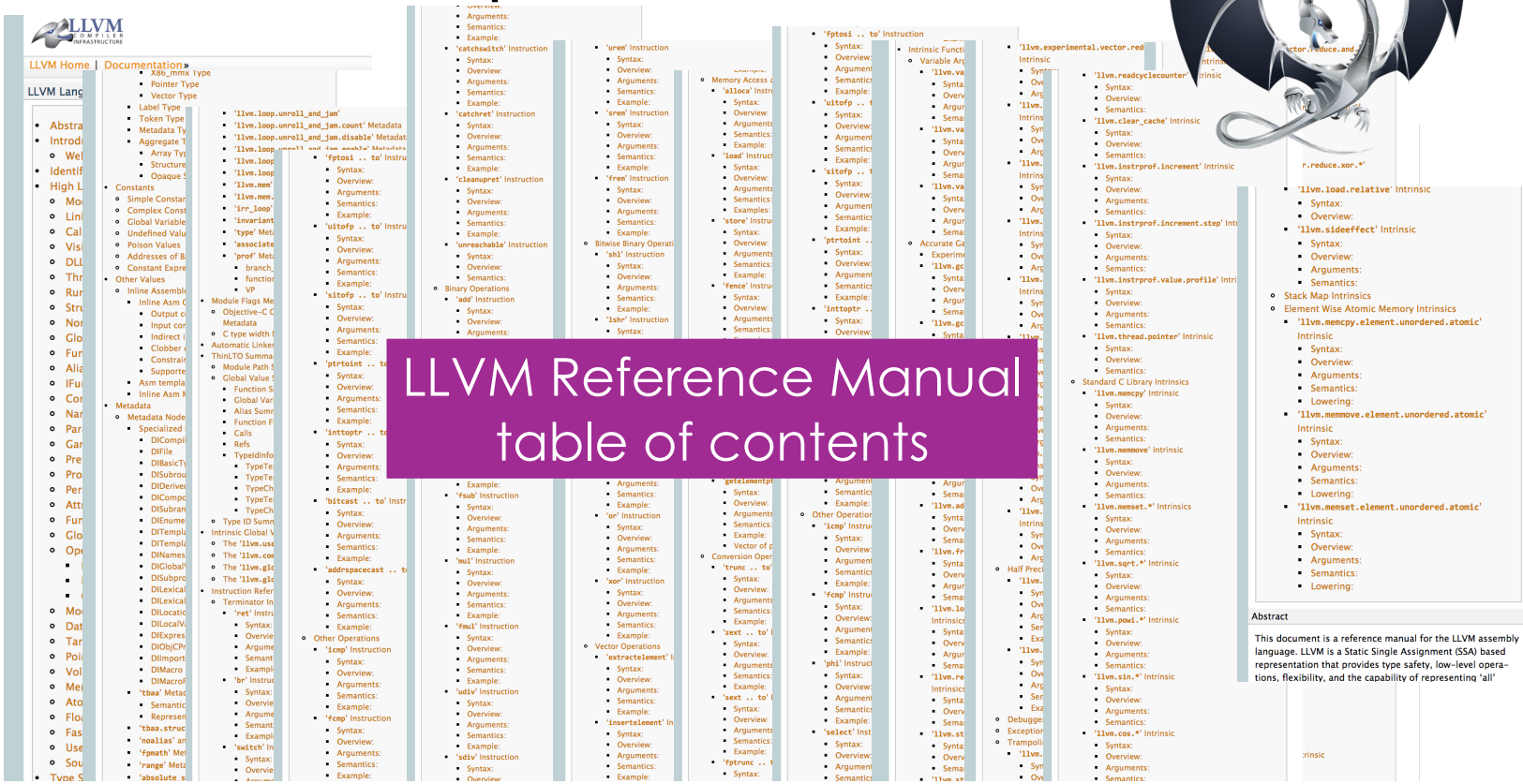
```
define i64 @factorial(i64 %n){
 %acc = alloca i64
 store i64 1, i64* %acc
 br label %start

start:
  %n1 = phi i64 [%n, %0], [%n2, %then]
 %c  = icmp sgt i64 %n1, 0
 br i1 %c, label %then, label %end

then:
  %x1 = load i64, i64* %acc
  %x2 = mul i64 %x1, %n1
 store i64 %x2, i64* %acc
  %n2 = sub i64 %n1, 1
 br label %start

end:
 %ans = load i64, i64* %acc
 ret i64 %ans
}
```

But... it's complex



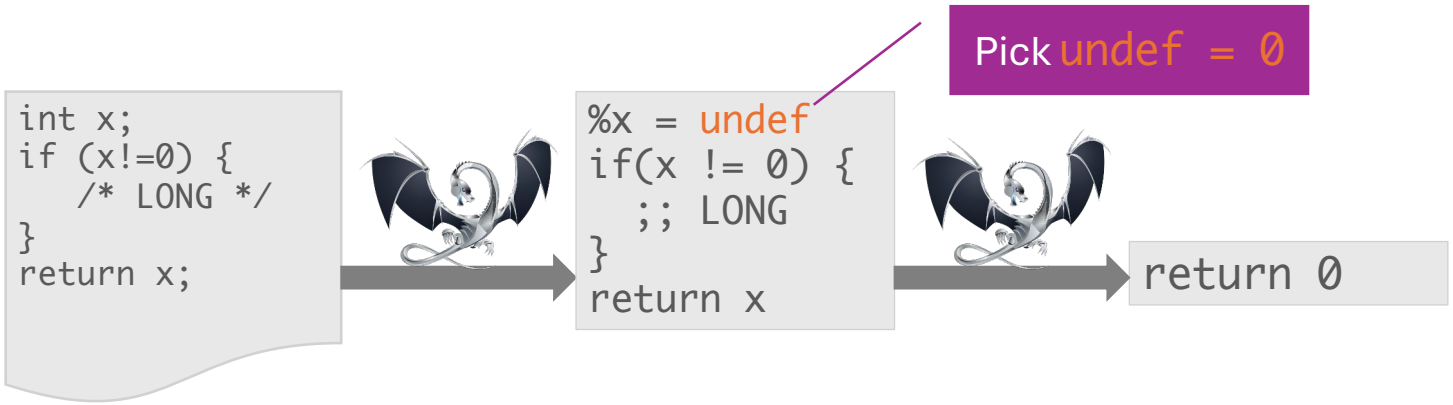
One Example: `undef`

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

Optimization with undef



(Slightly artificial example, branching on undef is UB in LLVM)

Optimistically use the best value
⇒ significantly faster code!

Interactions with Optimizations

Consider:

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

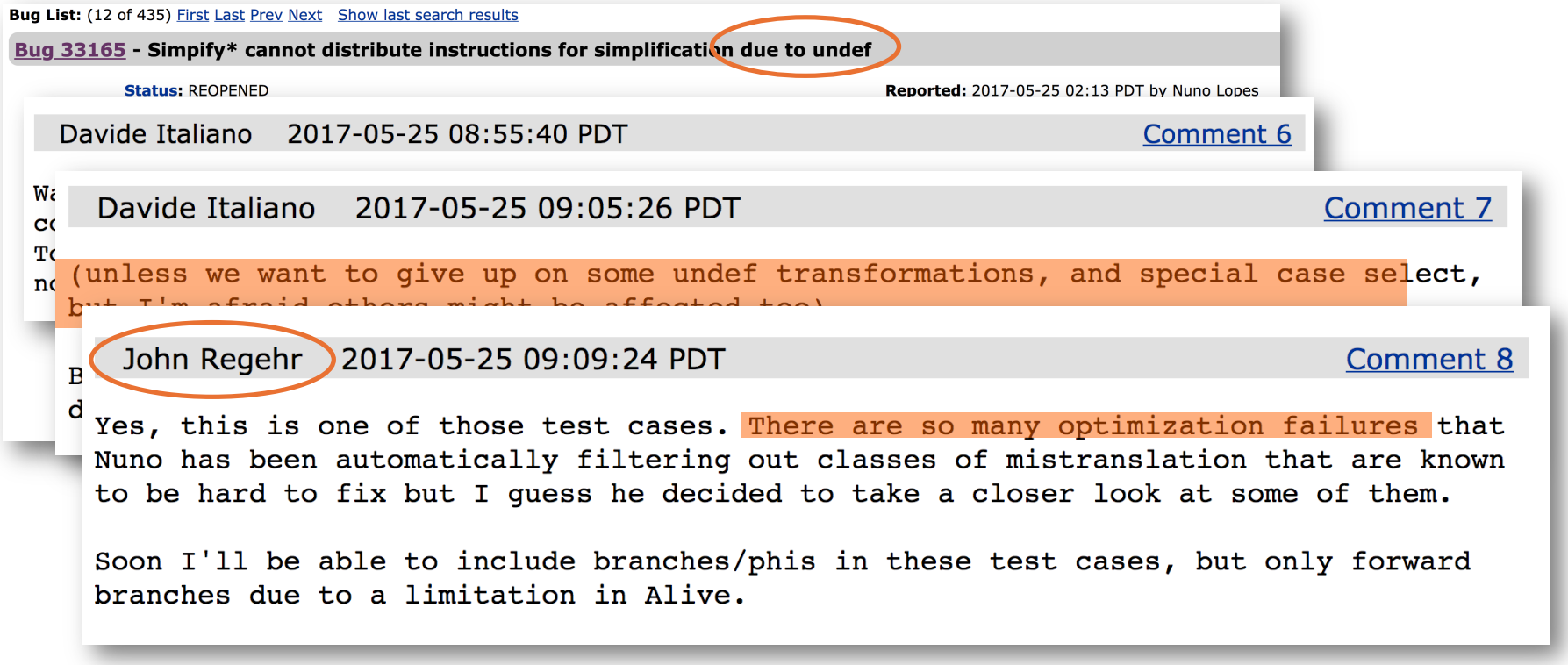
```
⟦%x⟧ = ⟦i8 undef⟧
     = {0,1,2,3,4,5,…,255}
⟦%y⟧ = {a mul 2 | a∈⟦%x⟧}
     = {0,2,4,…,254}
```

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}
```

What are the Challenges?



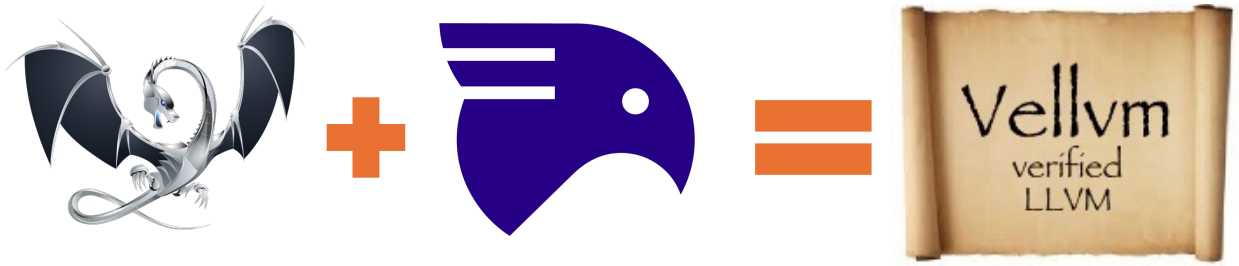
Outline

- LLVM
  - pain points: undef ptrtoint/ lack of intptr type, varargs, platform-specific undef + memory
- **Vellvm**
  - **Sketch of Formalization**
- Rocq Experience
- Differential Testing
  - GenLLVM structure
  - results
- Future

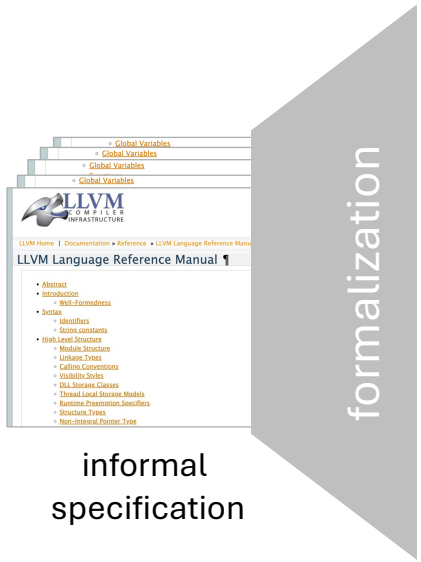
What Can We Do?

- LLVM is important
- But complex, features interact subtly
- Can lead to bugs :(

Maybe we can formalize it?

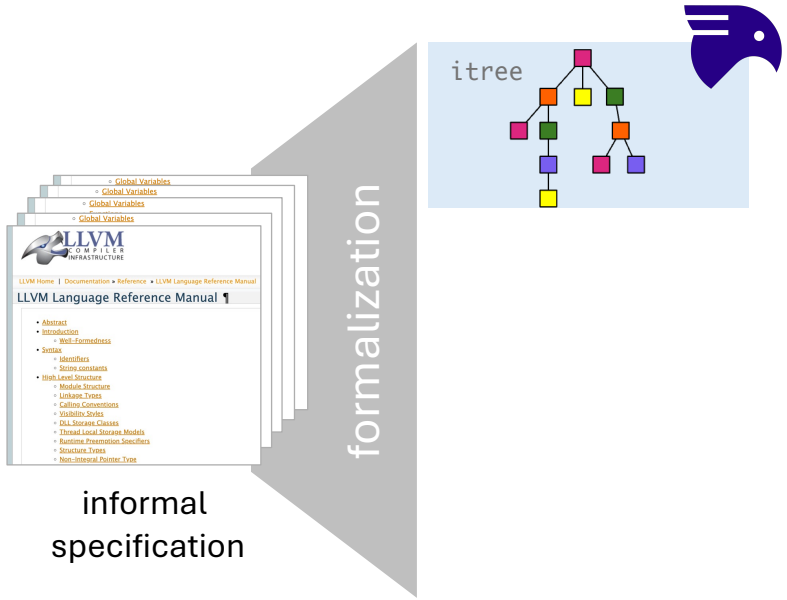


Structure of the Formalism: Informal Spec



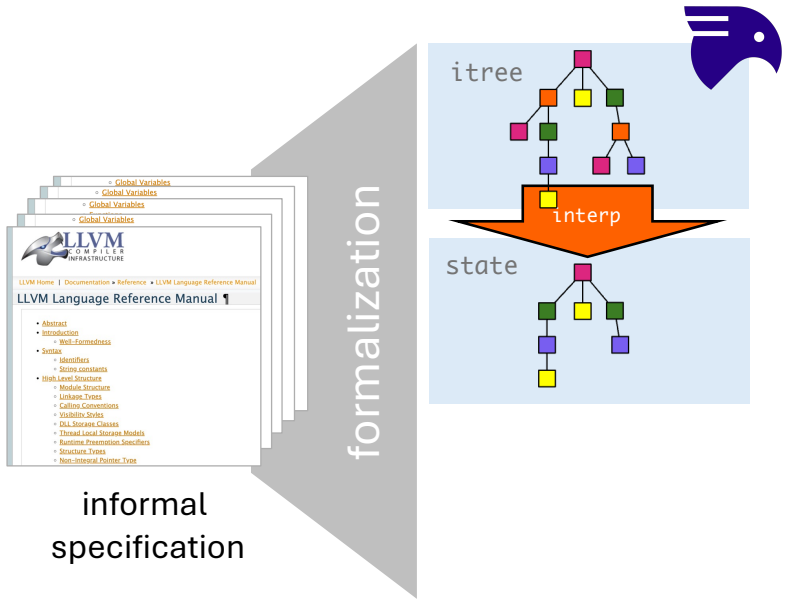
informal specification

Structure of the Formalism: Control Flow



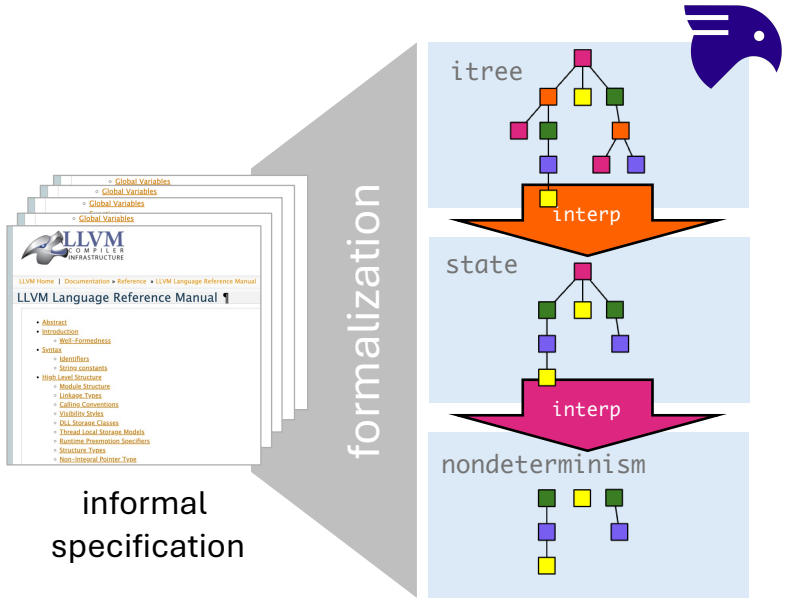
- Abstract Syntax
- Interaction Trees
  - "free monad" (nearly syntax)
  - infinitary behaviors

Structure of the Formalism: Monadic Interpreters

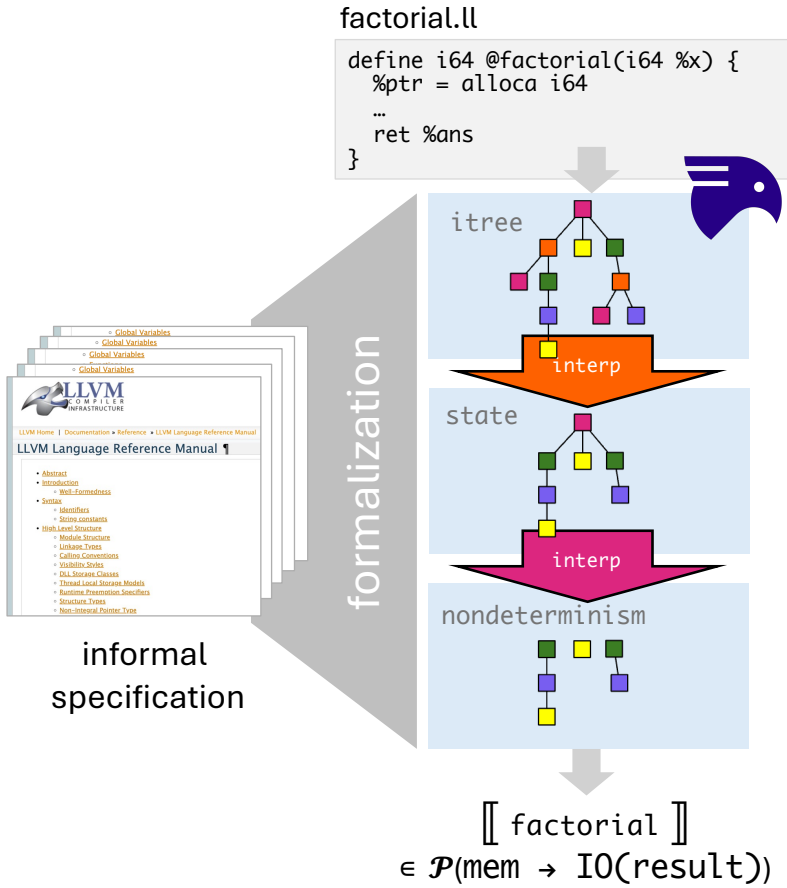


- Abstract Syntax
- Interaction Trees
  - "free monad" (nearly syntax)
  - infinitary behaviors
- Modular Event Handlers
  - Interpret the events
  - e.g., to introduce state

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.

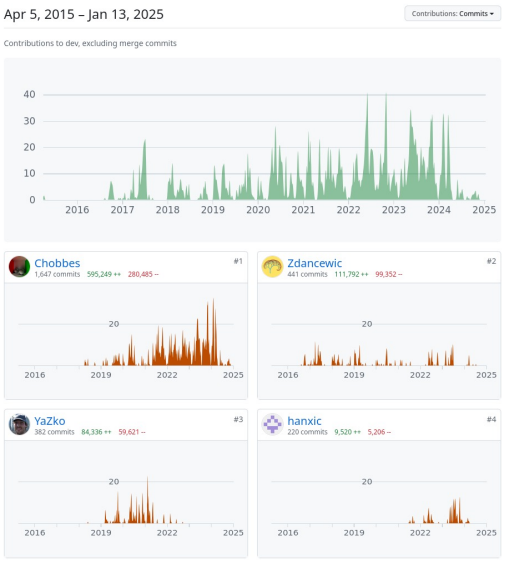
(Rough) Statistics about Vellvm Development

Lines of Spec	Lines of Proof
51.4K	95.6K

Size of vo files	Gzipped vo file size
516M	259M

Libraries:

- flocq
- paco
- itrees
- extlib
- quickchick
- (menhir)



Benefits of formalizing LLVM IR
Finding Bugs/Ambiguities

- Size 0 type issues
- sub-byte-sized pointer indices
- address guessing
  - "Reconciling high-level optimizations and low-level code in LLVM" [OOPSLA 18]
- lack of intptr_t (see ICFP 24)
- undef "entanglement" (see ICFP 24)
- ...

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?

Outline

- LLVM
  - pain points: undef ptrtoint/ lack of intptr type, varargs, platform-specific undef + memory
- Vellvm
  - Sketch of Formalization
- **Rocq Experience**
- Differential Testing
  - GenLLVM structure
  - results
- Future

Experience

Pros:

- Consistent rich logic
- Great for finding bugs / ambiguities
- Modules can be great
- Extraction (works shockingly well)

Cons:

- Constraints (Coinductive + existentials)
- Module issues
- More first class module types?
- Extraction bugs
- Standard Library
- Compilation time
- Labor intensive

Rocq Experience: Coinductive Existentials

- Arises due to nondeterminism
- Many constraints.

Rocq Experience: Coinductive Existentials

- Arises due to nondeterminism
- Many constraints.

```
a : itree A X
b : itree B X
b2 : itree B2 X
R : a ≋ b           (* Coinductive *)
MODEL : b2 ∈ modelB b
------------------------------------------
∃ (a2 : itree A2 X),
    a2 ∈ modelA a ∧
    a2 ≋ b2.
```

https://github.com/Chobbes/existential-coinduction-experiments

Outline

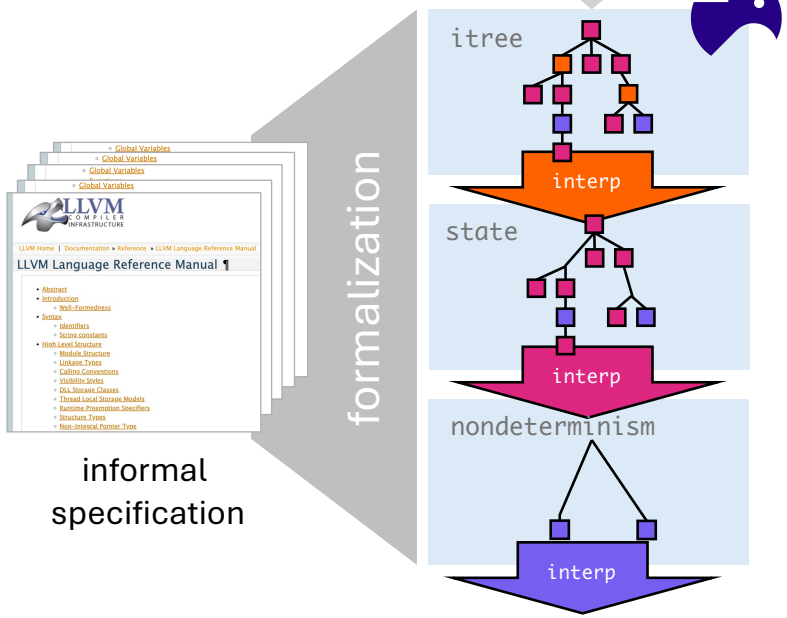
- LLVM
  - pain points: undef ptrtoint/ lack of intptr type, varargs, platform-specific undef + memory
- Vellvm
  - Sketch of Formalization
- Rocq Experience
- **Differential Testing**
  - **GenLLVM structure**
  - **results**
- Future

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

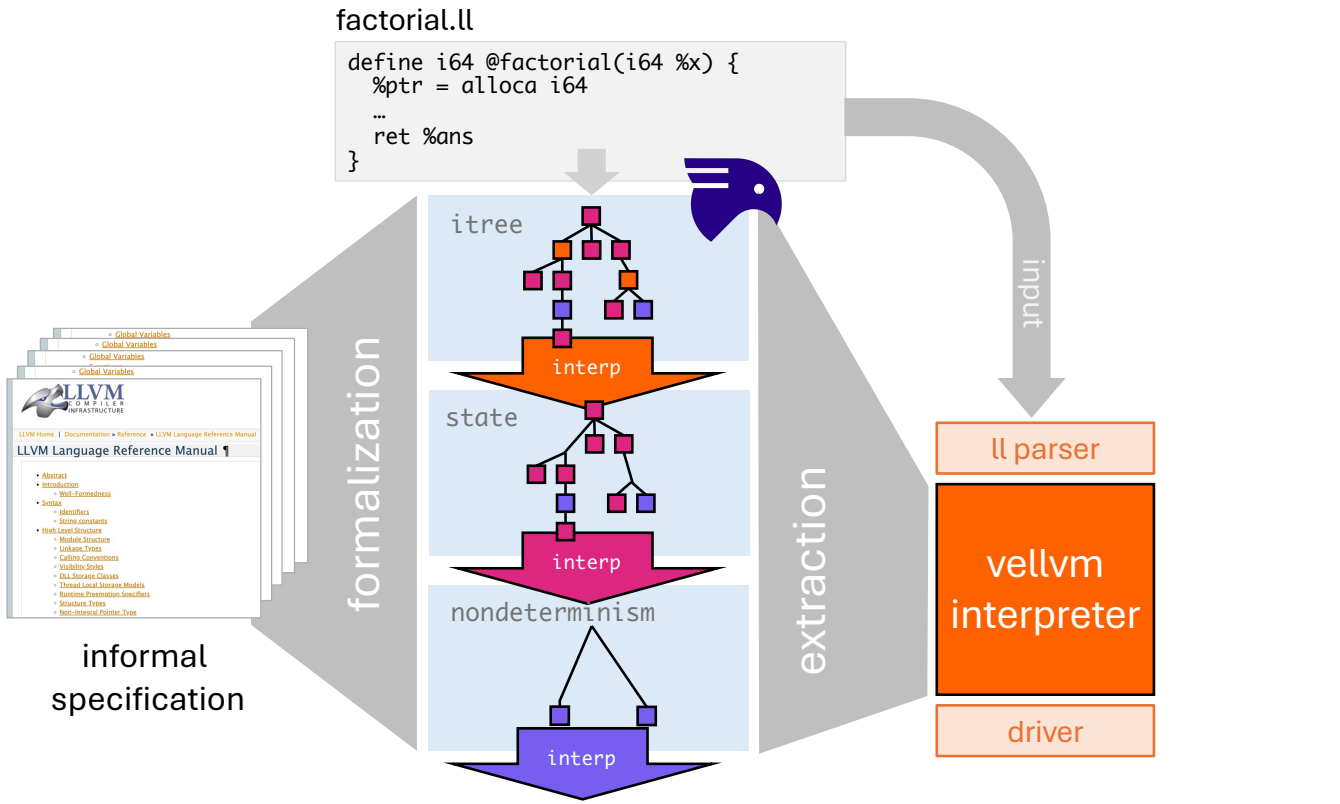
```
define i64 @factorial(i64 %x) {
  %ptr = alloca i64
  …
  ret %ans
}
```



informal specification

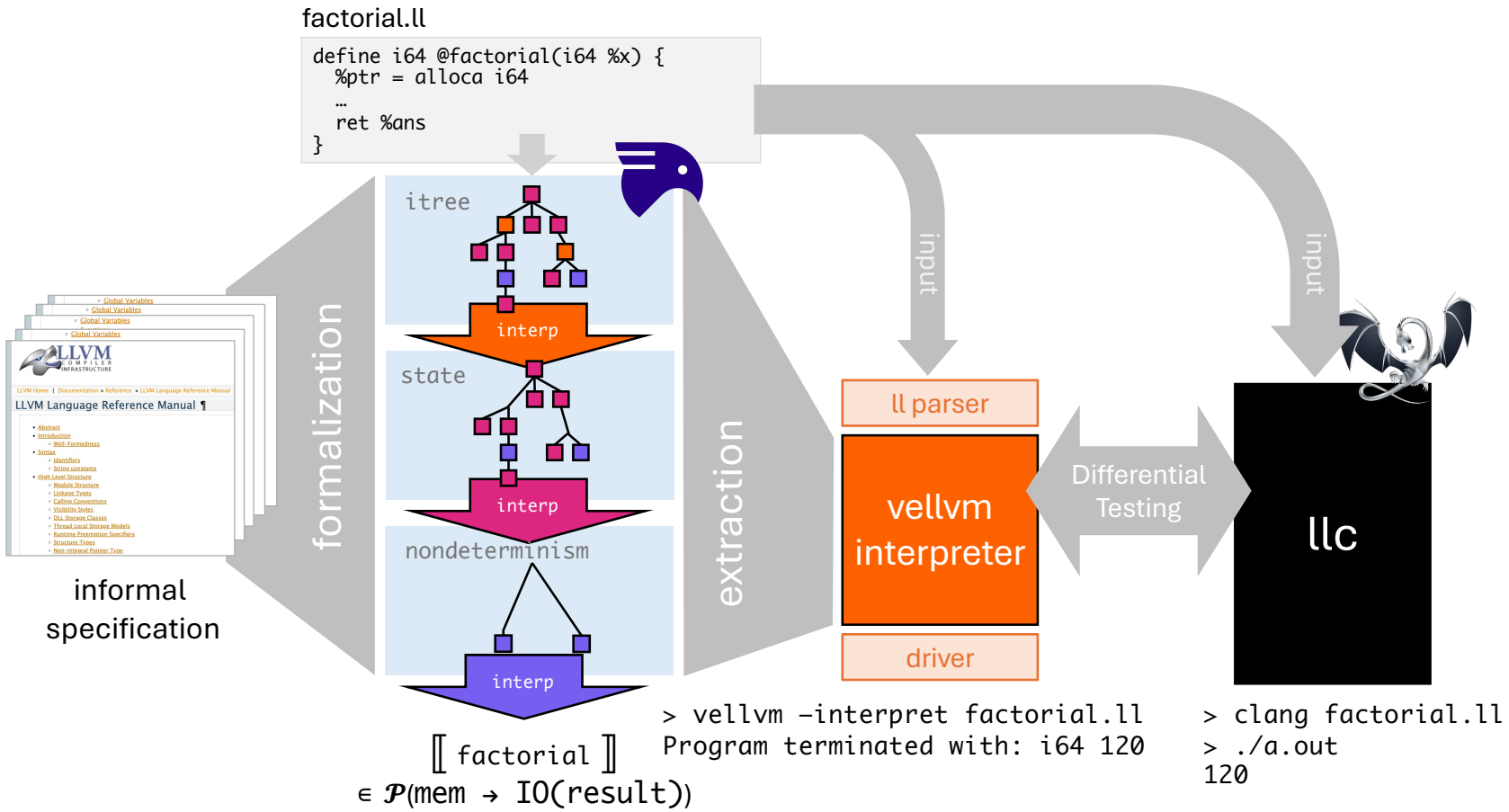
$[\![\texttt{factorial}]\!]$

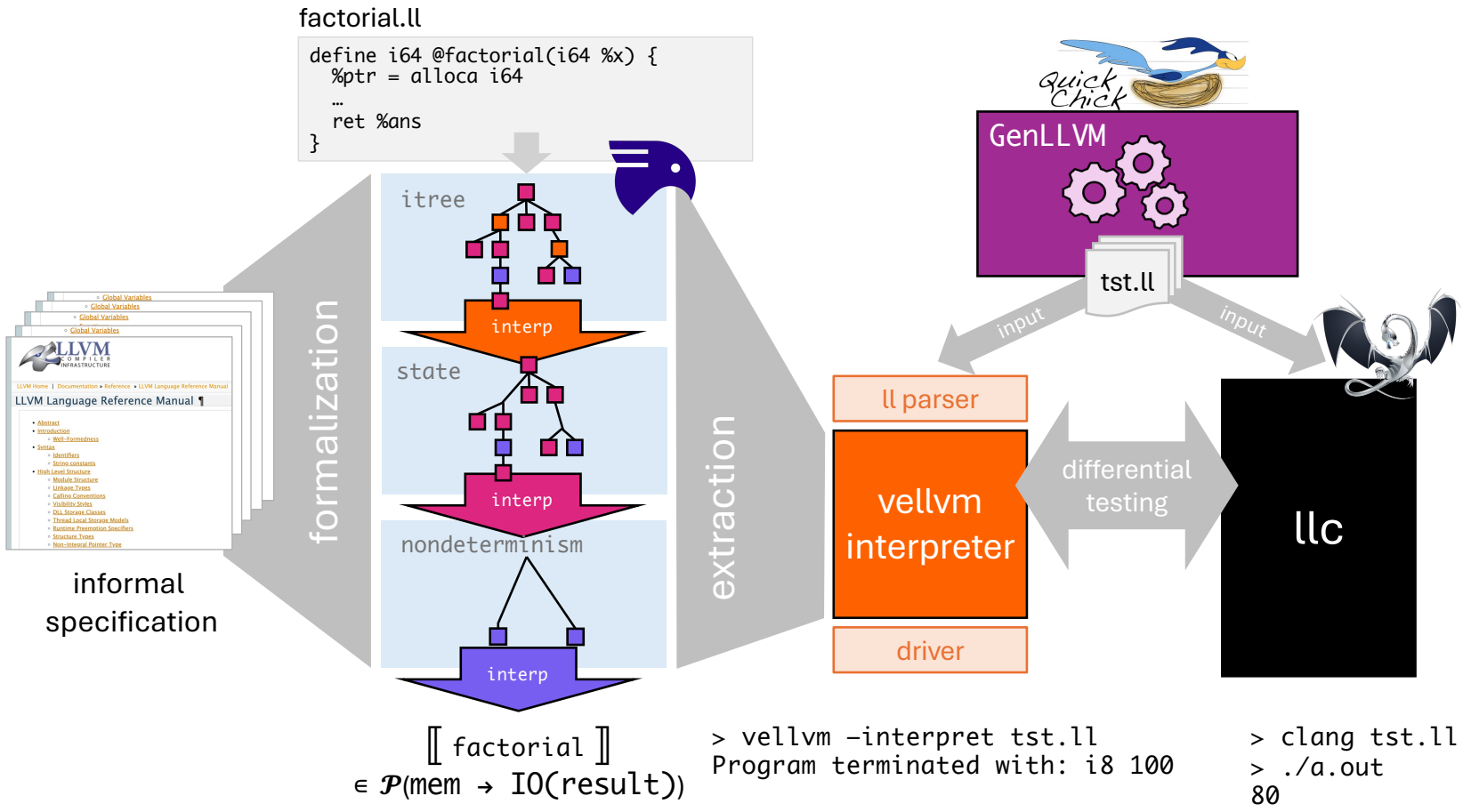
$\in \mathcal{P}(\texttt{mem} \to \texttt{IO(result)})$



⟦ factorial ⟧
∈ 𝒫(mem → IO(result))

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


factorial.ll
define i64 @factorial(i64 %x) {
%ptr = alloca i64
…
ret %ans
}
itree
interp
state
interp
nondeterminism
interp
formalization
extraction
input
input
LLVM Language Reference Manual ¶
informal specification
ll parser
vellvm interpreter
driver
Differential Testing
llc
⟦ factorial ⟧
∈ 𝒫(mem → IO(result))
> vellvm -interpret factorial.ll
Program terminated with: i64 120
> clang factorial.ll
> ./a.out
120


factorial.ll
define i64 @factorial(i64 %x) {
%ptr = alloca i64
…
ret %ans
}
itree
interp
state
interp
nondeterminism
interp
formalization
extraction
LLVM Language Reference Manual
informal specification
GenLLVM
tst.ll
input
input
ll parser
vellvm interpreter
driver
differential testing
llc
⟦ factorial ⟧
∈ 𝒫(mem → IO(result))
> vellvm -interpret tst.ll
Program terminated with: i8 100
> clang tst.ll
> ./a.out
80

Trickiness in Differential Testing of LLVM IR

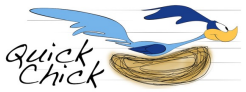
Requirements for LLVM Programs

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

Desirable Properties for Testing

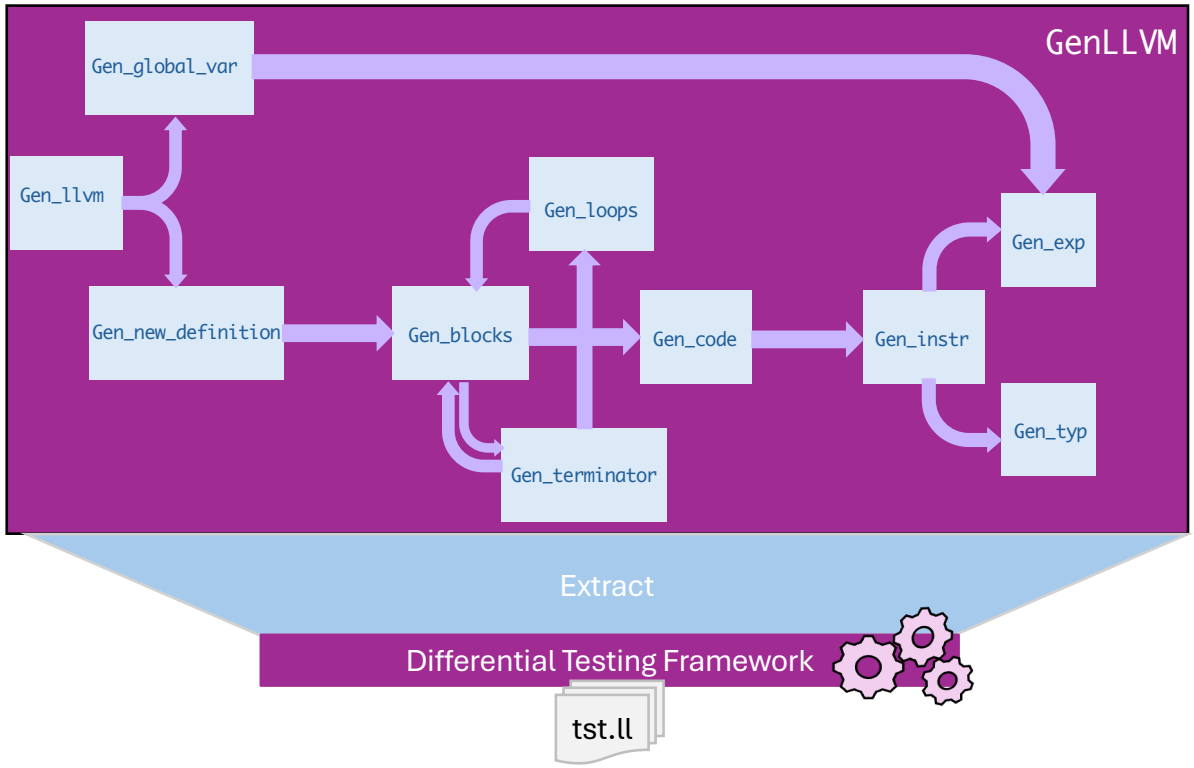
- Exercise large subset of LLVM syntax
- Nontrivial control flows
- Random code patterns

GenLLVM: Infrastructure



- Built using the G monad from the QuickChick library
- The GenLLVM monad can
  - Early termination when generating ill-formed code
  - Use a context-aware metadata system (inspired by Ecstacy)
  - Generate types / expressions / terminators

GenLLVM: Generation Strategy



Algorithm 1

procedure GEN_TERMINATOR(t, $back\_blocks$, $SIZE$) ▷ Generate an LLVM Function ▷ Each ϵ_i is a tunable frequency
  $r \leftarrow RANDOM(0, \epsilon_1 + \epsilon_2 + \epsilon_3 + \epsilon_4 + \epsilon_5)$
  if $r \leq \epsilon_1 || SIZE = 0$ **then** ▷ Generate return
    return $(gen\_ret\ t)$
  else if $r \leq \epsilon_2$ **then** ▷ Simple branching
    $(b, blocks) \leftarrow gen\_blocks(t, back\_blocks, \frac{SIZE}{2})$
    return $(br_1\ t\ b, blocks)$
  else if $r \leq \epsilon_3$ **then** ▷ Conditional branching
    $(b_1, blocks_1) \leftarrow backtrack(gen\_blocks(t, back\_blocks, \frac{SIZE}{2}))$
    $(b_2, blocks_2) \leftarrow gen\_blocks(t, back\_blocks, \frac{SIZE}{2})$
    $c \leftarrow gen\_exp\_size(0, i1)$
    return $(br_2\ i1\ c\ b_1\ b2, blocks_1 + +blocks_2)$
  else if $r \leq \epsilon_4$ **then** ▷ Generate Loop
    $(t, b, blocks) \leftarrow gen\_loop(t, back\_blocks, \frac{SIZE}{2})$
    $b' \leftarrow$ add b to the head of $blocks$
    return (t, b')
  else if $r \leq \epsilon_5$ **then** ▷ Generate recursion
    $b \leftarrow one\_of(back\_blocks)$
    return $(br_1\ t\ b, [])$

Figure: Algorithm for generating terminator

```
          gen_typ  :  GenLLVM typ
          gen_exp  :  typ → GenLLVM exp
        gen_instr  :  GenLLVM (list instr_id × instr)
         gen_code  :  GenLLVM (list instr_id × instr)
          gen_ret  :  typ → GenLLVM terminator
   gen_terminator  :  typ → list block_id →
                      GenLLVM (terminator × list block)
       gen_blocks  :  typ → list block_id →
                      GenLLVM (block × list block)
         gen_loop  :  typ → list block_id →
                      ℕ → GenLLVM (block × list block)
gen_new_definition :  typ → list typ → GenLLVM definition
    gen_global_var :  GenLLVM global
          gen_llvm :  GenLLVM (list toplevel_entity)
```

Figure: Types of GenLLVM's functions

Some Trickiness: inttoptr

```
%x = add i32 123, 0
%y = inttoptr i32 %x to i8*
%z = load i8, i8* %y
ret %z
```

- Nondeterminism
- Csmith and YARPGen do not generate `inttoptr` (mostly)!
- Observation: `inttoptr` *can* be invoked on integers that from `ptrtoint`!
  - (following size restrictions)

Evaluation

✓✓: emphasized; ✓: generated;
~: constrained; × : not generated

Feature	Csmith	YARPGen V1	YARPGen V2	GenLLVM
Level	C	C / C++	C / C++	LLVM
Int Arith	✓✓ (wrappers)	✓	✓	✓
Float Arith	✓	~ (constrained)	~ (constrained)	~ (LLVM Semantics)
Loops	✓ (induction var.)	✓	✓✓	✓ (induction var.)
Vectors	×	~ (short vectors)	✓✓	✓
Agg. Data	✓ (C)	✓ (C)	✓ (C)	✓✓
Ptr. Casts	✓ (only to void*, C)	×	×	✓

- Ran Vellvm on 10,000 programs each from GenLLVM, Csmith, YARPGen versions 1 and 2
  - (extracted) Vellvm can execute 2.5M+ LLVM IR instructions in 2m03s.
  - ✓: generated; ~: constrained; × : not generated

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!

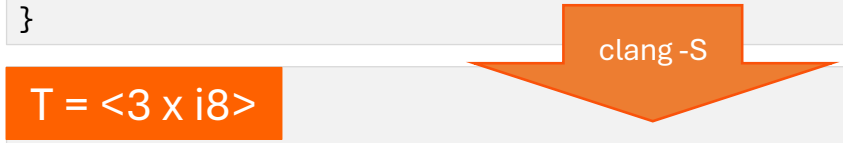
But:

```
clang: 46

Vellvm: 0
```

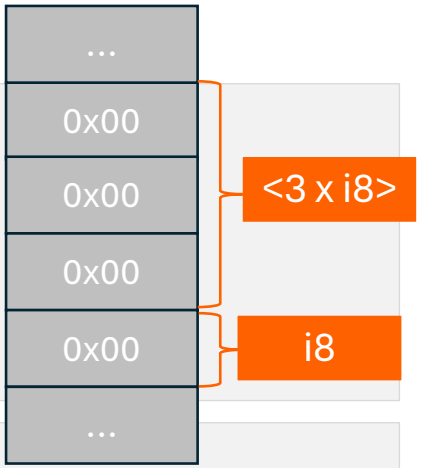
We Found Bugs in Clang!

```
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
}
```



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

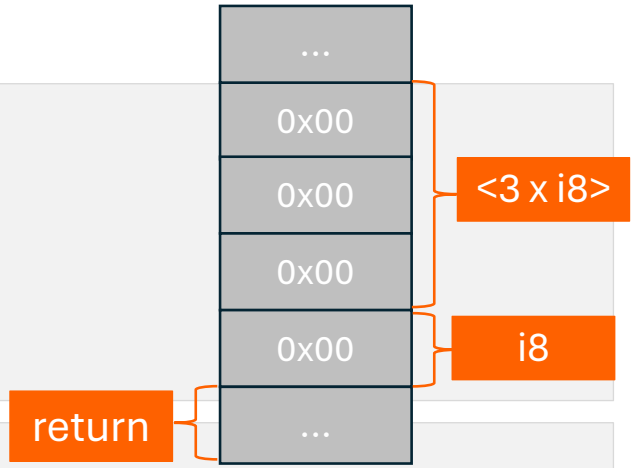
We Found Bugs in Clang!

```
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

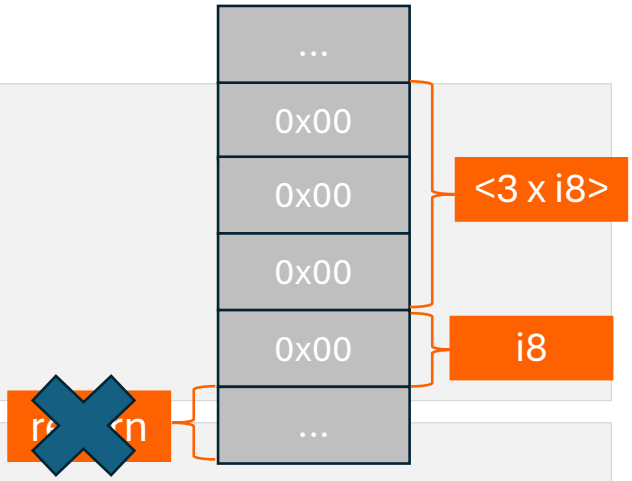
We Found Bugs in Clang!

```
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>

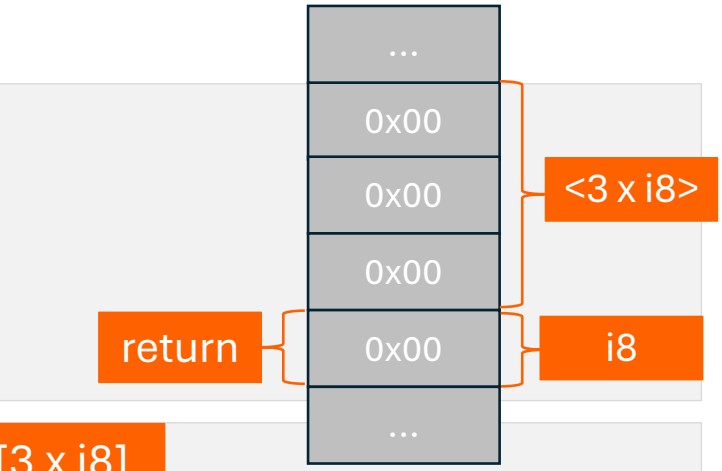
```
main:
    addi    sp, sp, -16
    mv      a0, zero
    sw      a0, 12(sp)
    lb      a0, 16(sp)
    addi    sp, sp, 16
    ret
```



- 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

Takeaways

- Rocq is great for formalizing complex systems!
- Coinduction & ITrees work well for defining semantics
- Executability & differential testing were essential
  - for debugging our semantics
  - for finding bugs in Clang & Langref

Together: these make Vellvm a useful platform for understanding LLVM IR semantics

Wrap-up / 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

Appendix

Rocq Experience: Extraction

- Shockingly Robust
- Nice features like Extraction Blacklist, etc.
- A little buggy at times
  - Modules?
  - Things named 'int'?

Rocq Experience: Modules

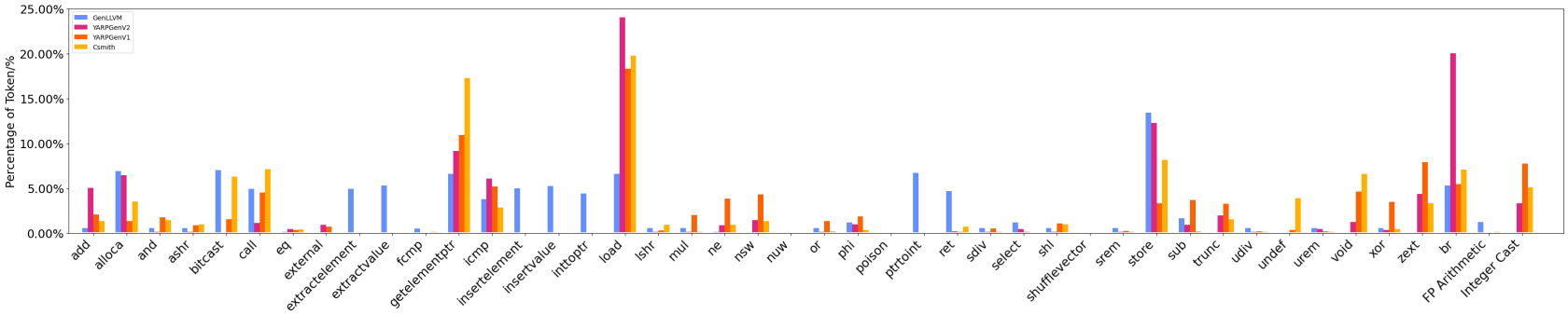
- Modules / module types are fantastic!
- Wish they were a bit more first-class
  - Construct anonymous module types with (<+) operator?
- Careful with Program in Module Types…

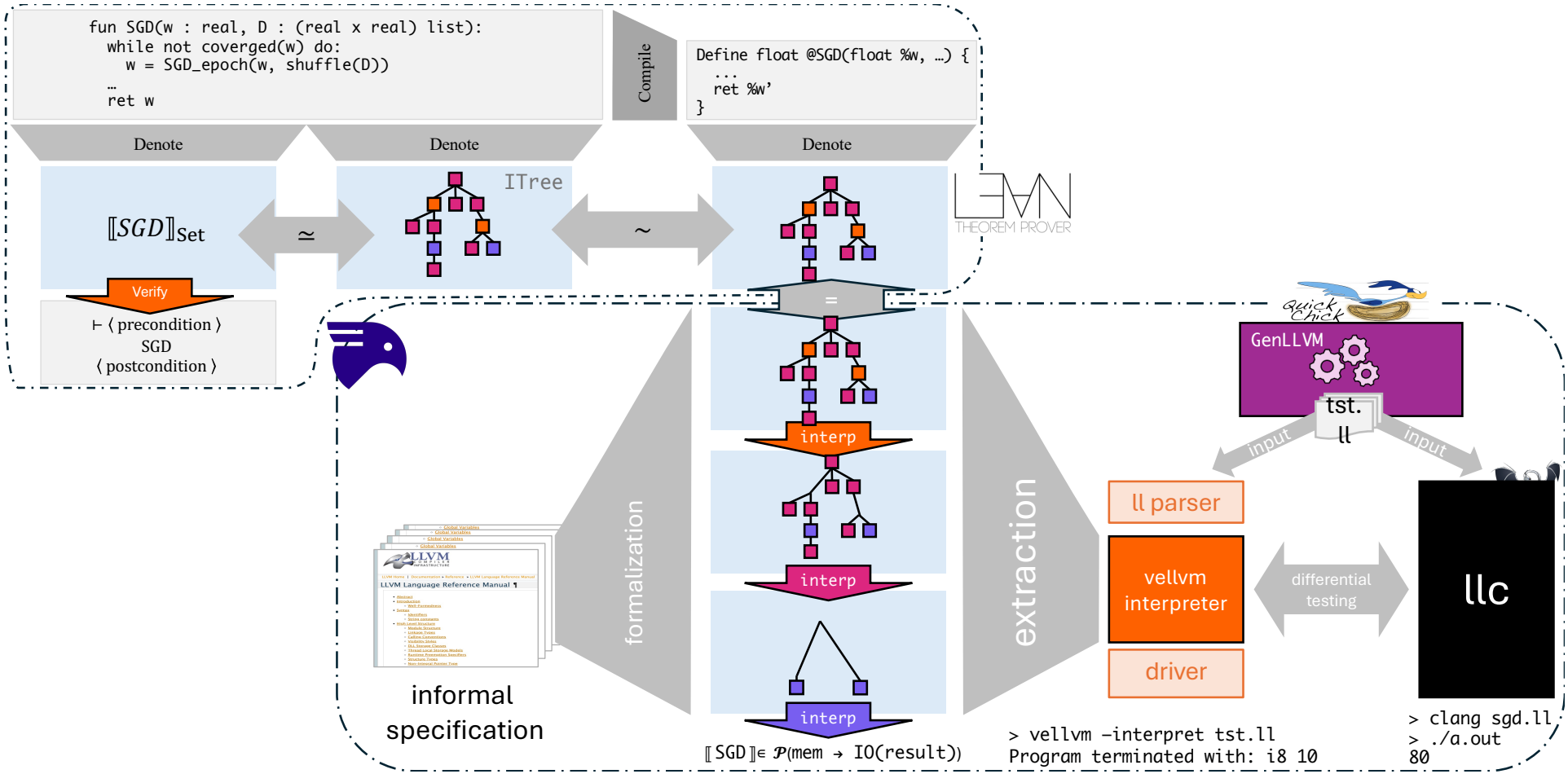
Undef Interactions with Memory

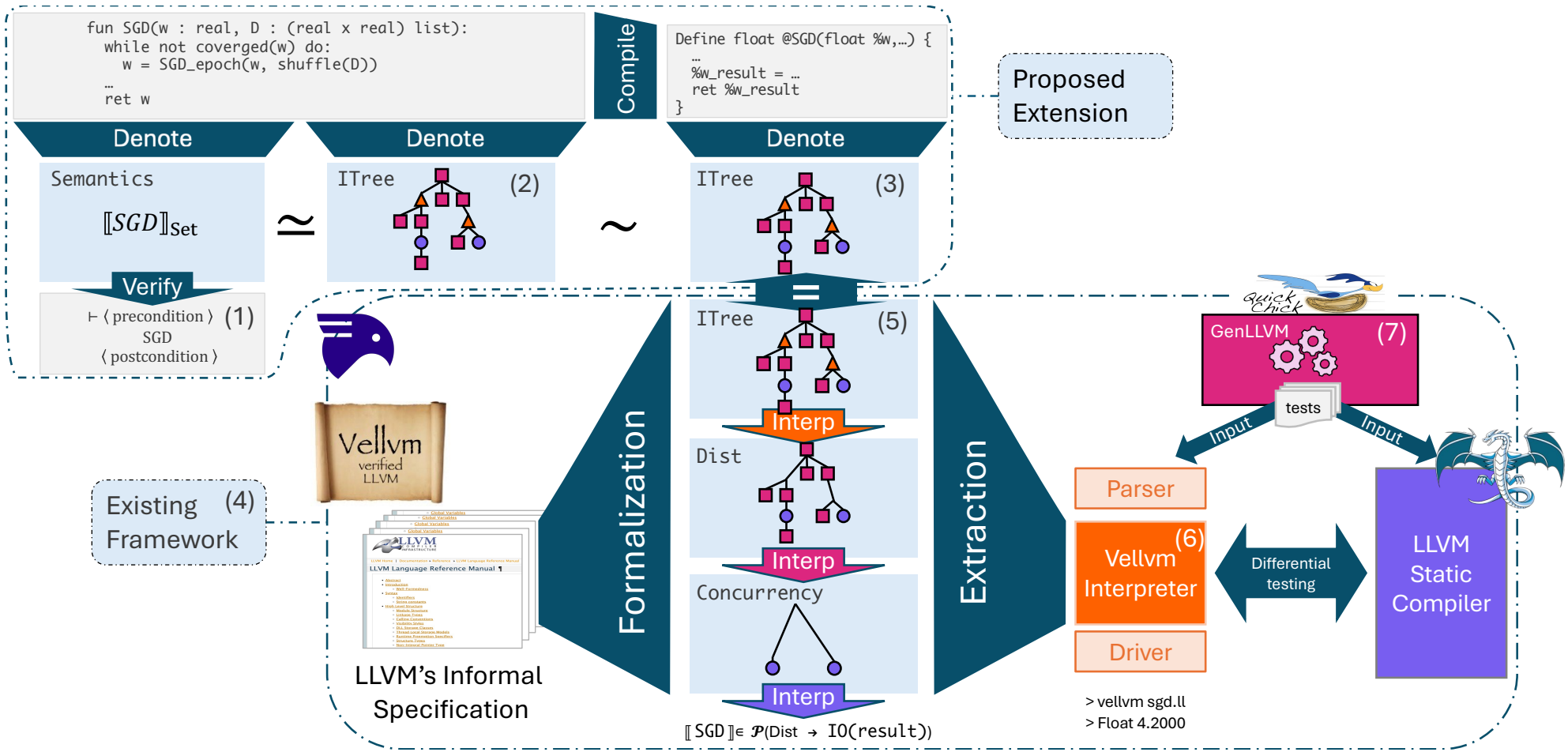
Important for optimizations like *store forwarding*

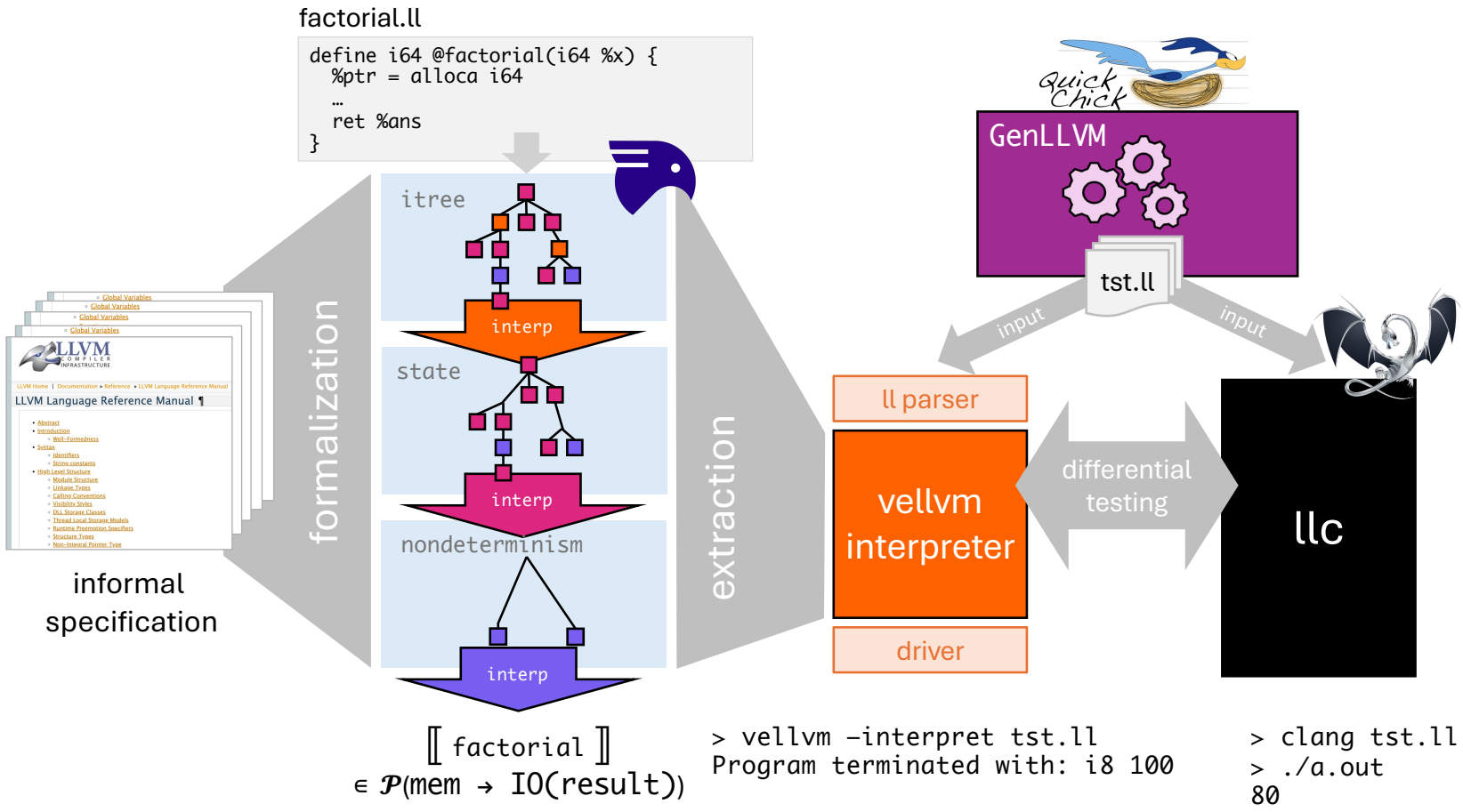
```
%x = i16 select i1 undef, i16 0x1234, i16 0x0000
%y = i16 select i1 undef, i16 0x5678, i16 0x0000
store i16 %x, ptr %ptr
store i16 %y, ptr (inttoptr (add iptr (ptrtoint %ptr) 2)
%z = load i32, ptr %ptr
```

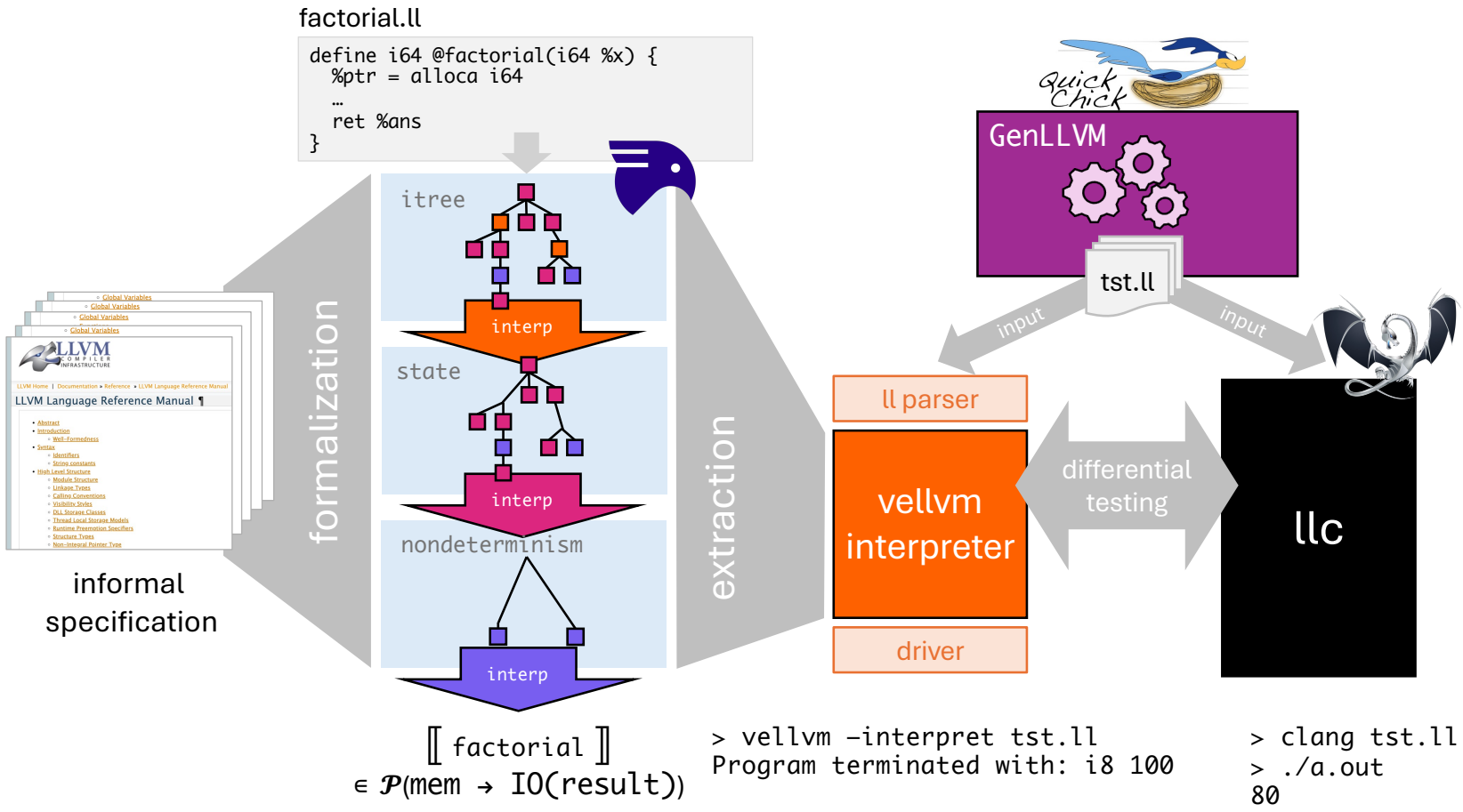
Comparison of Instructions Exercised




fun SGD(w : real, D : (real x real) list):
while not coverged(w) do:
w = SGD_epoch(w, shuffle(D))
…
ret w
Compile
Define float @SGD(float %w, …) {
...
ret %w'
}
Denote
Denote
Denote
⟦SGD⟧Set
≃
ITree
~
THEOREM PROVER
Verify
⊢ ⟨ precondition ⟩
SGD
⟨ postcondition ⟩
=
interp
interp
interp
formalization
extraction
informal
specification
⟦SGD⟧∈ 𝒫(mem → IO(result))
GenLLVM
tst.
ll
input
input
ll parser
vellvm
interpreter
driver
differential
testing
llc
> vellvm -interpret tst.ll
Program terminated with: i8 10
> clang sgd.ll
> ./a.out
80


fun SGD(w : real, D : (real x real) list):
while not coverged(w) do:
w = SGD_epoch(w, shuffle(D))
…
ret w
Compile
Define float @SGD(float %w,…) {
…
%w_result = …
ret %w_result
}
Proposed Extension
Denote
Denote
Denote
Semantics
$[\![SGD]\!]_{\text{Set}}$
≃
ITree
(2)
∼
ITree
(3)
Verify
⊢ ⟨ precondition ⟩ (1)
SGD
⟨ postcondition ⟩
ITree
(5)
Interp
Dist
Interp
Concurrency
Interp
[SGD] ∈ 𝒫(Dist → IO(result))
Formalization
Extraction
Vellvm
verified
LLVM
Existing Framework (4)
LLVM's Informal Specification
GenLLVM
(7)
tests
Input
Input
Parser
Vellvm Interpreter
(6)
Driver
Differential testing
LLVM Static Compiler
> vellvm sgd.ll
> Float 4.2000


factorial.ll
define i64 @factorial(i64 %x) {
%ptr = alloca i64
…
ret %ans
}
itree
interp
state
interp
nondeterminism
interp
formalization
extraction
LLVM Language Reference Manual
informal specification
GenLLVM
tst.ll
input
input
ll parser
vellvm interpreter
driver
differential testing
llc
⟦ factorial ⟧
∈ 𝒫(mem → IO(result))
> vellvm -interpret tst.ll
Program terminated with: i8 100
> clang tst.ll
> ./a.out
80


factorial.ll
define i64 @factorial(i64 %x) {
%ptr = alloca i64
…
ret %ans
}
itree
interp
state
interp
nondeterminism
interp
formalization
extraction
LLVM Language Reference Manual
informal specification
GenLLVM
tst.ll
input
input
ll parser
vellvm interpreter
driver
differential testing
llc
⟦ factorial ⟧
∈ 𝒫(mem → IO(result))
> vellvm -interpret tst.ll
Program terminated with: i8 100
> clang tst.ll
> ./a.out
80