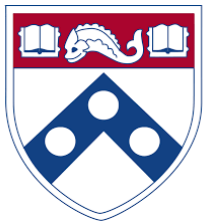


Verified LLVM IR



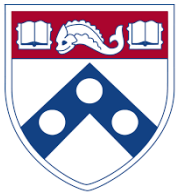
Steve Zdancewic
University of Pennsylvania
2026

Goal: Get your feedback about how Vellvm might be useful for this community?

https://vellvm.github.io/vellvm

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 Santo
Roger Burtonpatel
Santosh Nagarakatte
Gregory Malecha
Benjamin Pierce
Christine Rizkallah
Lucas Silver
Li-yao Xia
Irene Yoon
Vadim Zaliva
Yannick Zakowski
Jianzhou Zhao



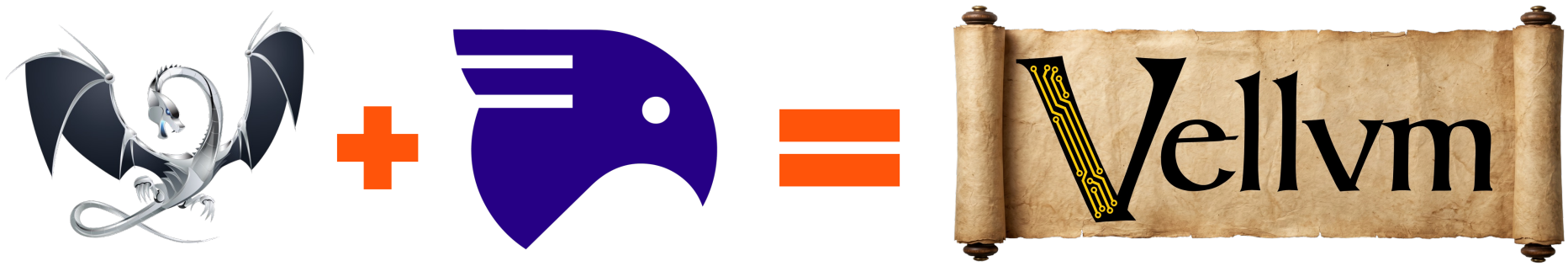






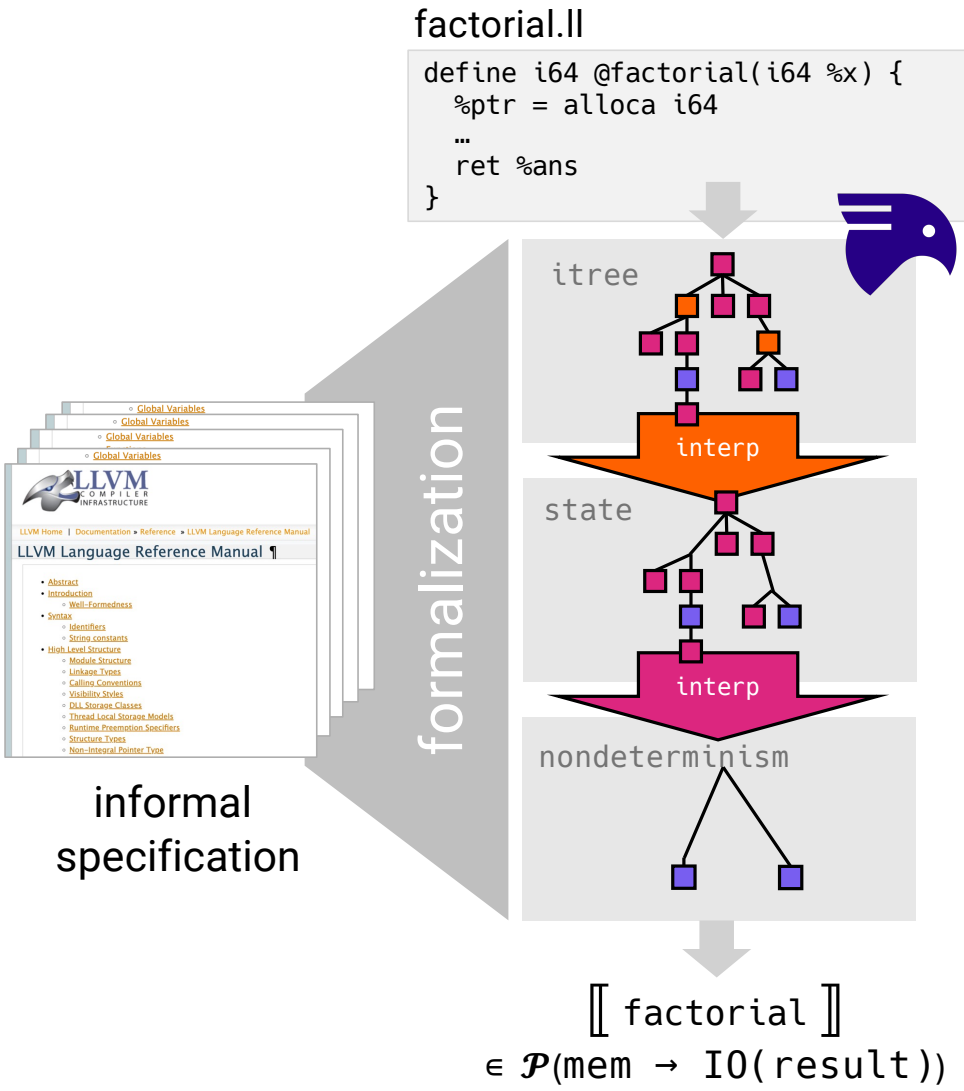






Vellvm is an LLVM IR semantic model implemented in Rocq

Rocq is an *interactive theorem* prover (like Lean)



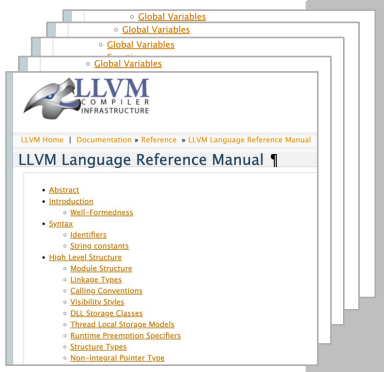
What is Vellvm?

1. **Rocq Specification of LLVM IR**
   - defines *sets* of allowed behaviors
     - captures *nondeterminism*
   - executable interpreter
2. **Definition(s) of *refinement***
   - what does it mean for a program transformation to be correct?
3. **Infrastructure for proofs**
   - for probing the semantics
   - for translation validation
   - for verifying properties of compiler passes

Executable Interpreter

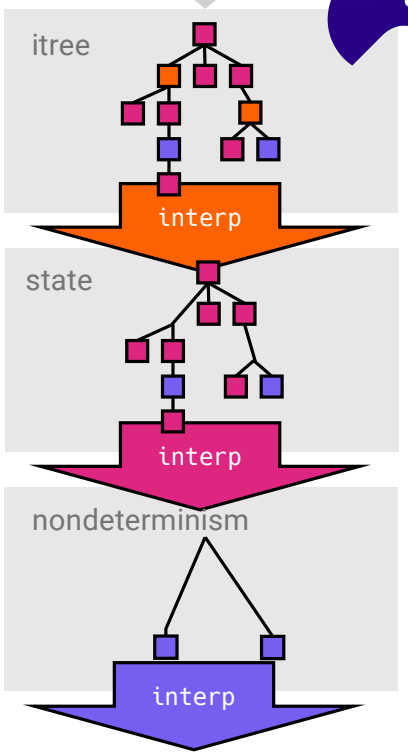
factorial.ll

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



informal specification

formalization



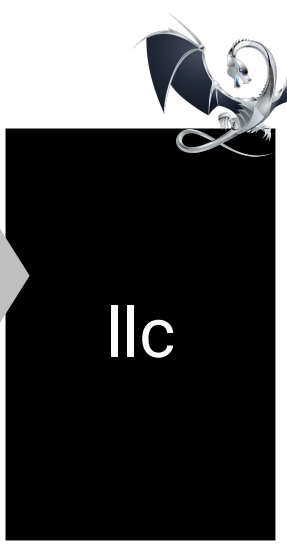
extraction

ll parser

vellvm interpreter

driver

differential testing



⟦ factorial ⟧
∈ 𝒫(mem → IO(result))

```
> vellvm -interpret tst.ll
Program terminated with: i8 120
```

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

Vellvm Demo

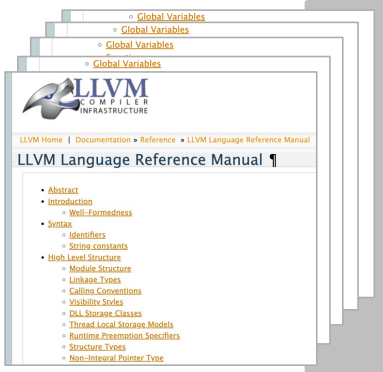
- Hello World
  - UB
- make test
  - Executable tests: e.g. tests/ll/aggregate-layout.ll
  - Comparison with llubi
- debugger

So What?

- **Vehicle for research**
  - how to build complex / large specifications in Rocq?
  - experiment with language design
- **Finding bugs in clang**
  - thinking hard about corner cases
  - identify inconsistent assumptions of the LLVM compiler
  - run in tandem with with CSmith / YarpGen
  - Complementary to Alive2?
- **Verified compilers a la CompCert**
  - e.g., Helix front end → Vellvm, CertiRocq → Vellvm (in progress)
  - is a particular optimization correct?
  - does the target assembly refine the source?
- **Verifiably correct programs**
  - program logics require a semantics
  - machine-checked compliance with specifications

factorial.ll

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



informal specification

formalization

itree

interp

state

interp

nondeterminism

interp

extraction

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

l parser

vellvm interpreter

driver

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

$\ni$

```
> vellvm -interpret tst.ll
Program terminated with: i8 120
```

EXAMPLE Semantics Experiment (TAKE WITH GRAIN OF SALT!)

What changes if `undef` means `freeze poison` instead of `poison`?

Vellvm's interpreter under both semantics, every assertion in the test suite run alone; llubi for reference.

16 **of 3,846 assertions change outcome**

All are ASSERT_UB cases in `tests/ub` with `undef` in a position where poison is UB.

Assertions that hold

`undef = poison`	**3,812**
`undef = freeze poison`	**3,804**

The 8 lost are UB that is no longer reported.

Alive2 `@src` / `@tgt` functions
run with no expected result

differ, of 50 run	**13**
UB → a value	**6**
poison result → a value	**5**
UB → UB for another reason	**2**

The 16 assertions, by where the undef is used

		Vellvm `undef = poison`	Vellvm `undef = freeze poison`	llubi default and undef = 0
br / switch condition	3	UB	no UB (returns a value)	no UB
alloca element count	1	UB	no UB (returns a value)	no UB
noundef argument / return value	2	UB	no UB (returns a value)	no UB
memcpy / memset length	2	UB	no UB (returns a value)	UB
divisor of udiv / sdiv / urem / srem	5	UB	UB (division by zero)	no UB
load / store address	2	UB	UB (invalid address)	UB
indirectbr address	1	UB	UB (invalid target)	UB
Reports UB	**16**	**16 of 16**	**8 of 16**	**5 of 16**

reports undefined behavior runs to completion

Vellvm's `freeze` picks zero. Count column: number of assertions. Source: `tests/undef/compare.py`, 2026-11-16, LLVM 23.1.2.

vect_sum: clang -O2 refines clang -O0, machine-checked in Rocq + Vellvm

vect.c

```
for (int i = 0; i < len; i++)
  a[i] = b[i] + c[i];
```

clang -O0 → **vect-O0.ll** — scalar, 1 element / iteration

len, i, a, b, c live in 5 stack slots (alloca)

```
%16 = load i32, ptr %9          ; i
%19 = load i32, ptr %18         ; b[i]
%24 = load i32, ptr %23         ; c[i]
%25 = add nsw i32 %19, %24
store i32 %25, ptr %29          ; a[i]
```

≈ **vect_spec** (Spec.v)

the C loop as a pure function

`State → OK σ' | UB | error`

same operations as -O0, in order;
only ever OK or UB (vect_spec_okub)

⊒ clang -O2 → **vect-O2.ll** — no stack slots, vectorized

scalar only if len < 4 or a is within 64 B
after b or c (runtime alias check); else:

main loop · 16 / iter	`4 × add nsw <4 x i32>`
epilogue · 4 / iter	`add nsw <4 x i32>`
tail · 1 / iter	`add nsw i32`

O0.v · O0_function eutt between the -O0 run and the spec;
its stack slots are hidden (mem_ext) and popped on return

VectBody.v composes the two sides
spec UB ⇒ anything goes; poison args are UB (noundef)

O2.v, O2Loop.v · O2_function if the spec returns σ',
-O2 returns, with the same memory view as σ'

One run with len = 23 — when the alias check passes, one vector step = 4 spec steps, lane by lane (poison lanes included)

vect_spec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
-O2	**main loop, one iteration: 4 × <4 x i32>**																**epilogue: <4 x i32>**				i32	i32	i32

(with a within 64 B after b or c, -O2 runs only the tail loop: one step per spec step)

Precondition vect_pre on the caller's state and arguments (VectCall.v)

safe_args pointer arguments cannot reach blocks allocated later (else a caller could aim a at O0's stack slots)

typing arguments are i32, ptr, ptr, ptr

```
Theorem vect_O2_refines_O0 :
  refine_entry_pre vect_pre DTYPE_Void
    (Name "vect_sum") vect_O0 vect_O2.
```

RefineVect.v · every -O2 run is a legal behavior of the -O0 run

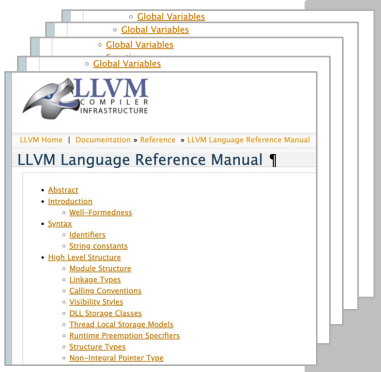
Example of "probing the semantics"

This lemma states that if you take an "abstract value" and serialize to the in-memory byte representation and then deserialize it, you get the same thing.

```
Lemma memory_bytes_round_trip₁ :
  ∀ dt dv ta mb,
    serializable dt →
    dvalue_has_dtyp dv dt →
    dvalue_to_memory_bytes dt dv ta = ret mb →
    memory_bytes_to_dvalue mb dt = ret dv.
Proof.
  intros dt dv ta mb HS HT HW.
  unfold dvalue_to_memory_bytes in HW.
  destruct (acc_dvalue_to_memory_bytes_h dt dv 0%N ta [])
    as [ | | | [off bs] ] eqn:E; cbn in HW; try discriminate.
  inversion HW; subst; clear HW.
  destruct (round_trip_h HT HS _ _ E) as (w & Ebs & _ & HR & _).
  rewrite Ebs, app_nil_r, rev_append_rev, app_nil_r.
  exact HR.
Qed.
```

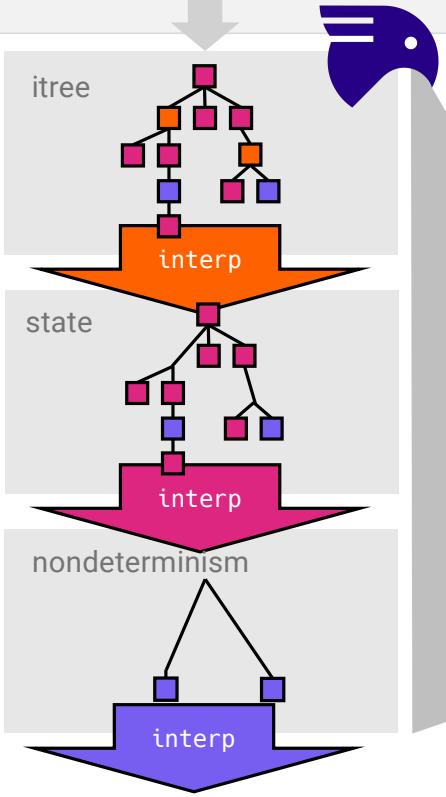
factorial.ll

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



informal specification

formalization



extraction

⟦ factorial ⟧
∈ 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 120
```

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

Evaluation

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

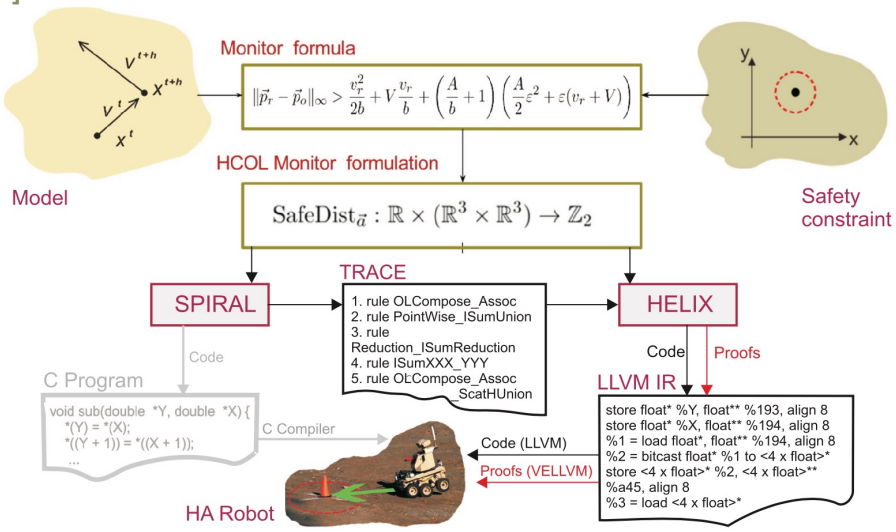
Feature	CSmith	YARPGen V1	YARPGen V2	GenLLVM
Level	C	C / C++	C / C++	LLVM
Int Arith	✓✓ (wrappers)	✓	✓	✓
Float Arith	✓	~ (constrained)	~ (constrained)	~ (LLVM Semantics)
Loops	✓ (induction var.)	✓	✓✓	✓ (induction var.)
Vectors	×	~ (short vectors)	✓✓	✓
Agg. Data	✓ (C)	✓ (C)	✓ (C)	✓✓
Ptr. Casts	✓ (only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

Spiral/Helix

[Püschel, et al. 2005,Franchetti et al., 2005, 2018, Zaliva et al., 2015 2018, 2019]

- DSL for High-performance numerical computations
- Compiles to LLVM IR
- Formalized in Rocq, targets Vellvm
- Bottom of the compilation chain proved





Caveats: Vellvm is not Complete!



- This is all a work in progress…
  - provenance model is WIP
  - semantic definition of many annotations (e.g., `noalias`)
  - no concurrency (yet)
  - missing many intrinsics
- LLVM LangRef "informal spec" unclear; target-dependent
  - exceptions (personality), inlined assembly
- Some simplifications (easily? solvable with engineering effort):
  - little-endian only, alignment (we ignore target datalayout)
  - only float and double (no half, bfloat, fp128, x86_fp80, ppc_fp128)
- Executable interpreter has limitations
  - Very large memory data structures
  - "Linking" against libraries is not possible (requires FFI)

Intentional Deviations from LLVM IR

(in the Vellvm main branch I demoed)

- no `undef`
  - anticipating removing it; undef keyword is treated like `poison`
- Memory model / semantics
  - `iptr` type (integer type like C's `intptr_t`, but unsigned)
  - padding is poison

- Vellvm-specific intrinsics
  - `@llvm.vellvm.internal.throw` -- raise an exception

What features of LLVM IR?

- Parser:
  - ~ clang version 23
- Vellvm assumes that the LLVM IR code is well formed
  - (no type checker / SSA invariant checker)
  - we trust that clang's checker is correct
- Semantics:
  - sequential instruction semantics
  - control flow & function calls
  - memory model (several variants)
    - byte type, per-bit poison a la Nuno's PLDI 2026 paper
    - Provenance model – in progress