Wrap-up Lecture

# CIS 5000: SOFTWARE FOUNDATIONS

Steve Zdancewic

Fall 2024

# Announcements

- HW12: Sub.v
  - due: Monday, Dec. 9th at ***10:00pm***

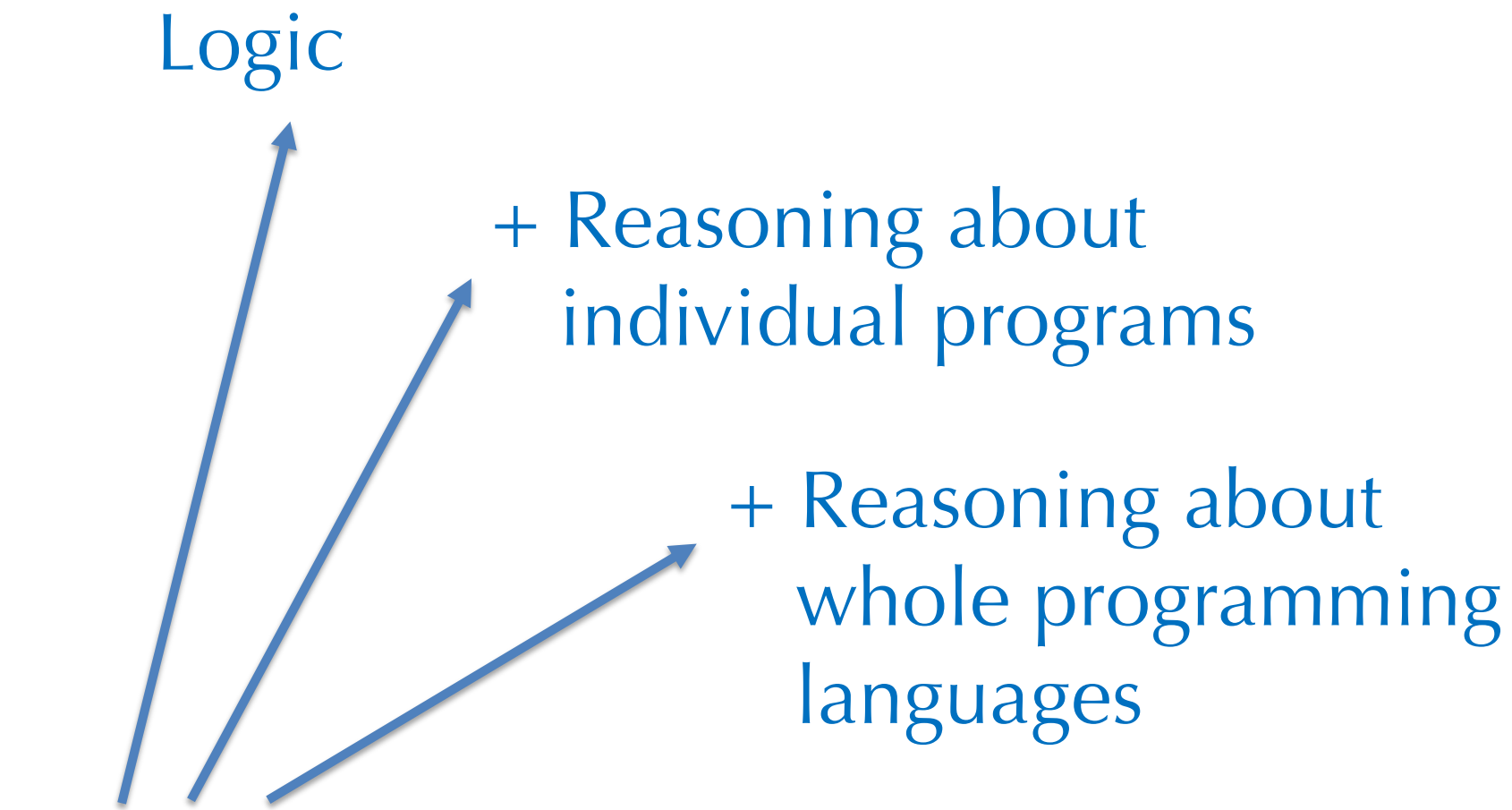
- **Final Exam:**
  **Thursday, December 12th noon – 2:00pm**
  **LRSM AUD**

# Final Exam Coverage

- The exam is comprehensive, but emphasizes more recent material:
- **General Coq principles:**
  - Propositions, induction, and inductively-defined propositions
  - Functional programming
- **Imp Semantics:**
  - large- and small-step operational semantics
  - program equivalence
  - Hoare logic
- **Simply-typed Lambda Calculus:**
  - operational semantics of higher-order functions, substitution
  - type system and proofs: canonical forms, preservation, progress
  - variants: products, sums, fix, records, etc.
- **Subtyping**
  - properties of the subtyping relation <:
  - inversion lemmas
- **Mutable state**
  - references, store typing (details less emphasized since there was no HW)

We will provide definitions, reminders, etc., so there is less emphasis on memorization.

# COURSE RECAP



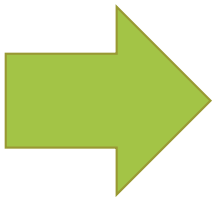

# SOFTWARE FOUNDATIONS

# CIS 5000

- **Foundations**
  - Functional programming
  - Constructive logic
  - Logical foundations
  - Proof techniques for inductive definitions
- **Semantics**
  - Operational semantics
  - Modeling imperative "While" programs
  - Hoare logic for reasoning about program correctness
- **Type Systems**
  - Simply typed λ-calculus
  - Type safety
  - Subtyping
  - Dependently-typed programming
- **Coq interactive theorem prover**
  - turns doing proofs & logic into programming → fun!

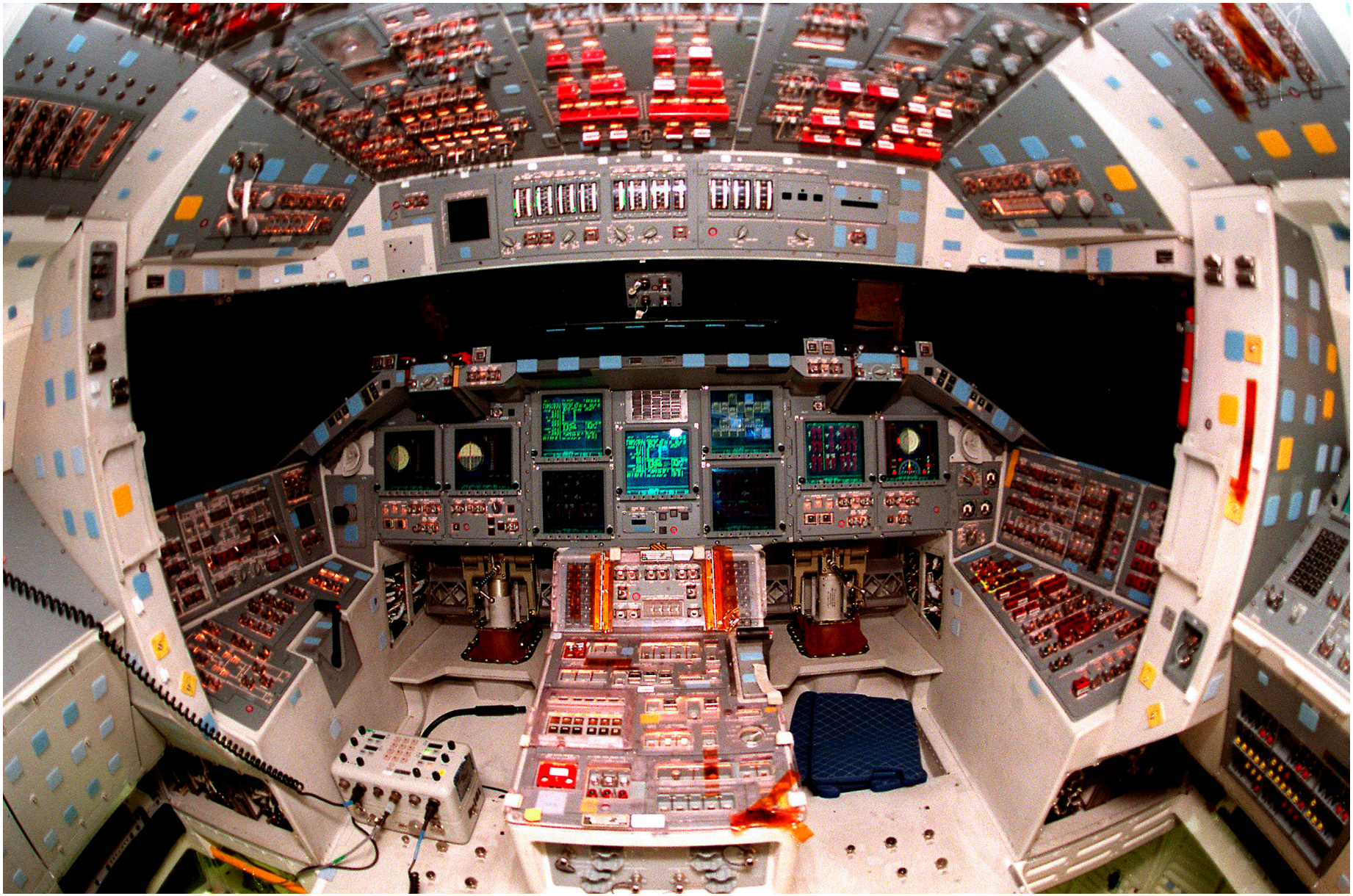
# Subset Used in CIS 5000

To start.




By the end of the semester.

# Coq's Full Capabilities



# Can Formal Methods Scale?

## Academia

- **Bedrock** – web; packet filters
- **CakeML** – SML compiler
- **CertiKOS** – certified OS kernel
- **CompCert** – C compiler
- **EasyCrypt** – crypto protocols
- **Kami** – RISCV architecture
- **HS2Coq** – Library validation
- **SEL4** – OS microkernel
- **Vellvm** – LLVM IR
- **VST** – C software
- **Ynot** – DBMS, web services

## Industry

AIRBUS

NASA

Microsoft

aws

Google

Meta

galois

# The Science of Deep Specifications



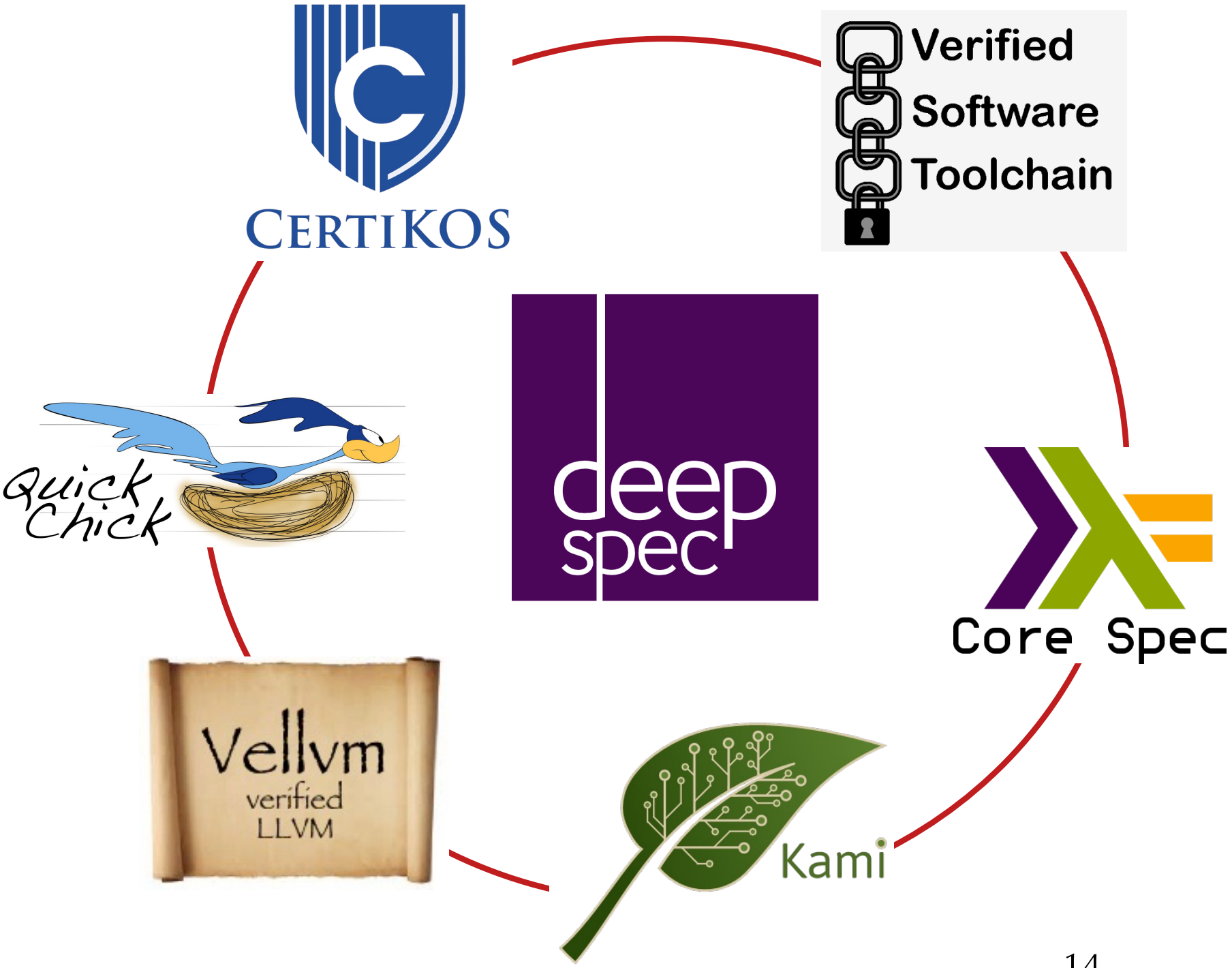
Expeditions Grant
deepspec.org

# 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).


CertiKOS
Verified
Software
Toolchain
deep
spec
Quick
Chick
Core Spec
Vellvm
verified
LLVM
Kami

case study

# SMART CONTRACTS

# Formal Methods for Blockchain

**Academic Work:**

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

CERTIK

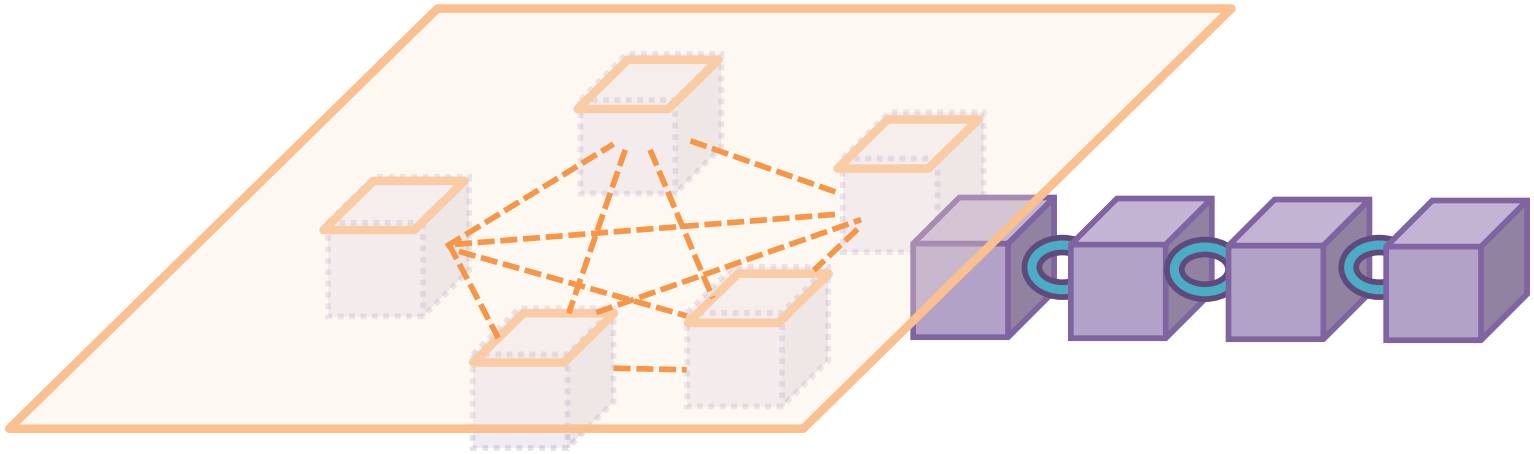
deep spec

Uses deep spec results

CARDANO

Tezos

# Smart Contracts

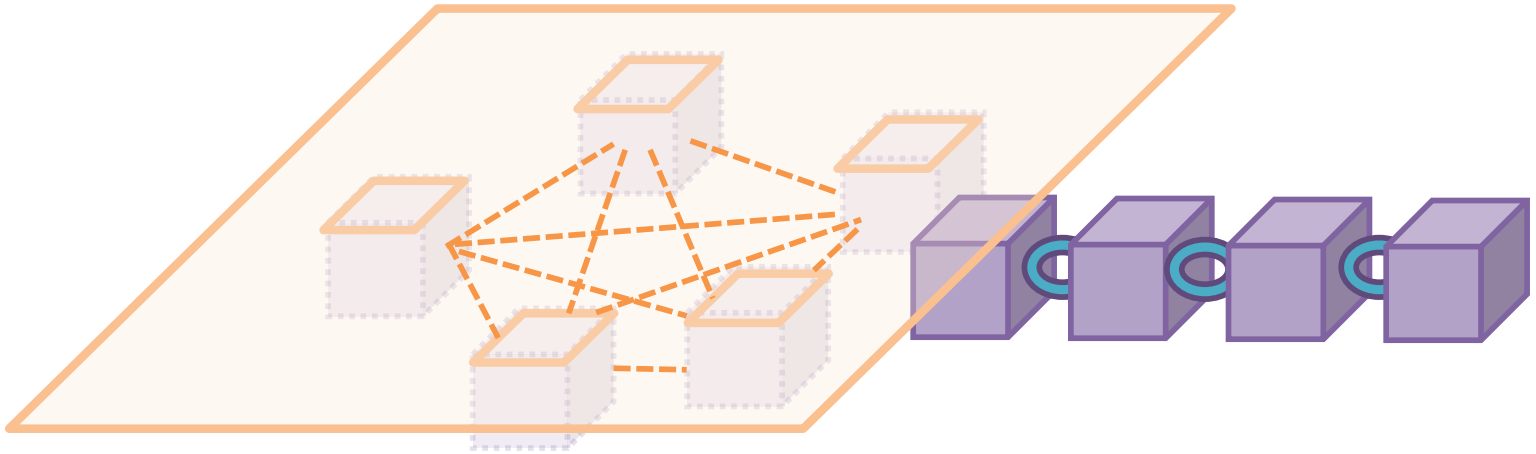


blockchain

# Smart Contracts

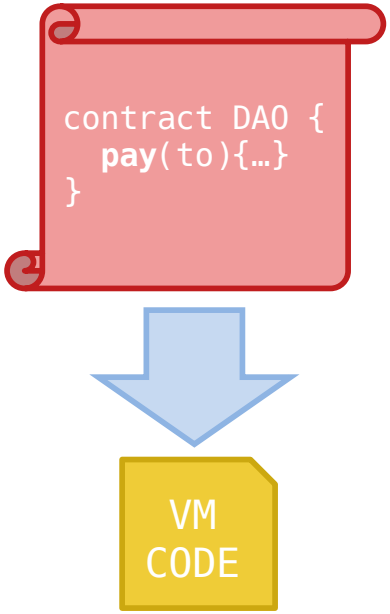

**smart contract**

- high-level
- human written
- programming language



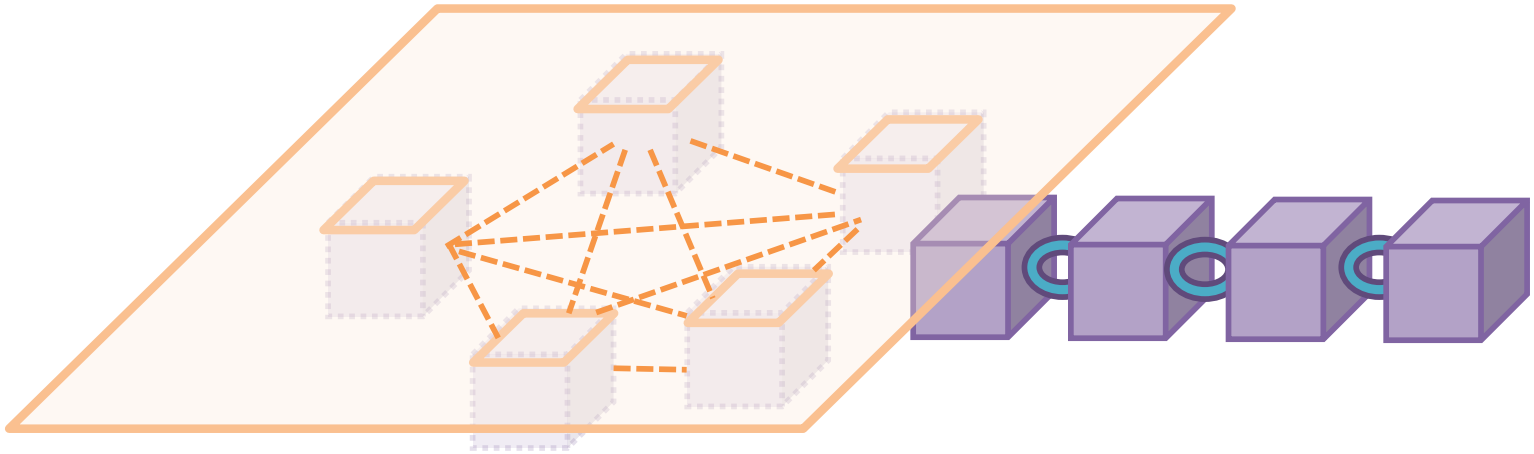
**blockchain**

# Smart Contracts



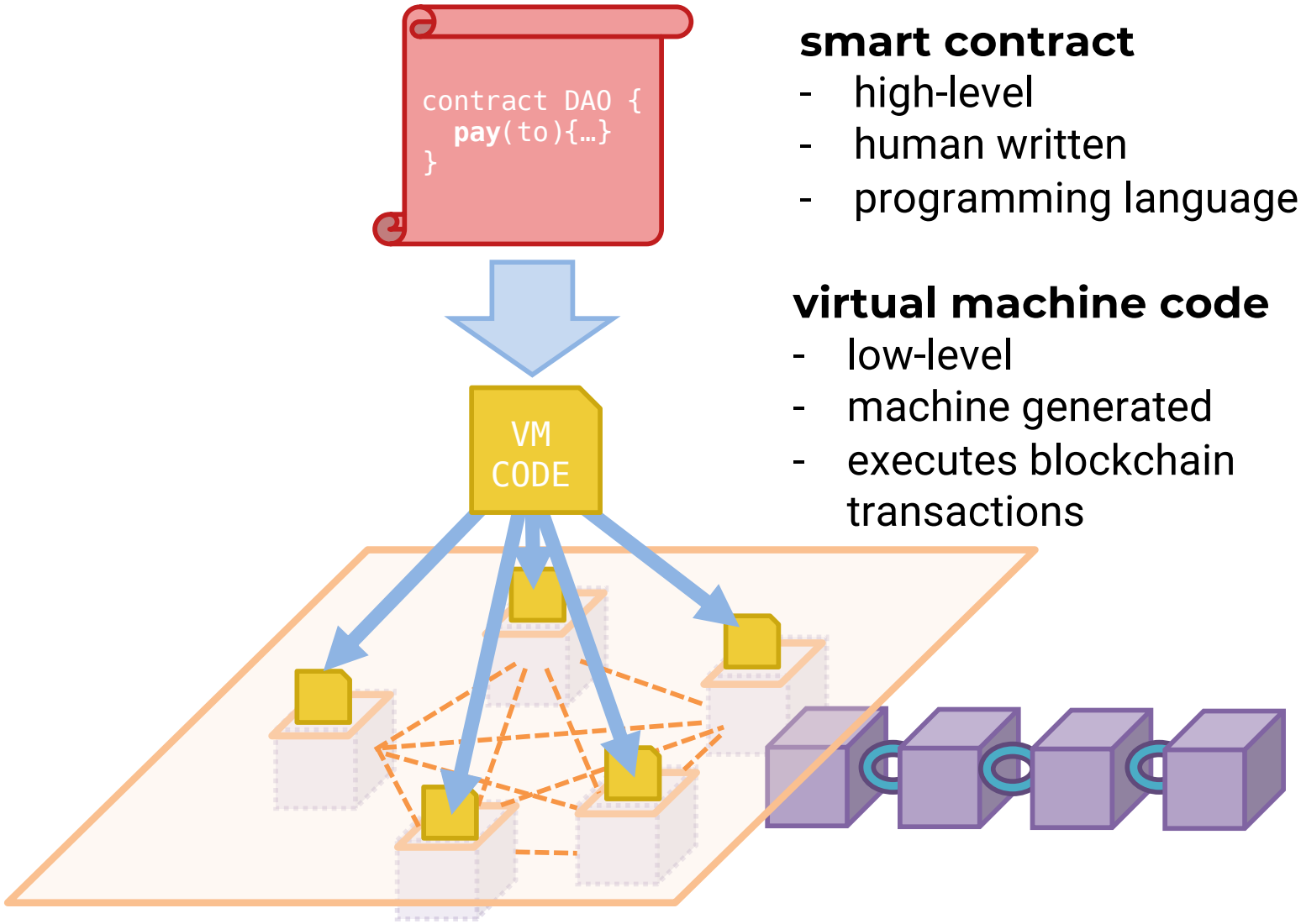

**smart contract**
- high-level
- human written
- programming language

**virtual machine code**
- low-level
- machine generated
- executes blockchain transactions



**blockchain**

# Smart Contracts




**smart contract**
- high-level
- human written
- programming language

**virtual machine code**
- low-level
- machine generated
- executes blockchain transactions

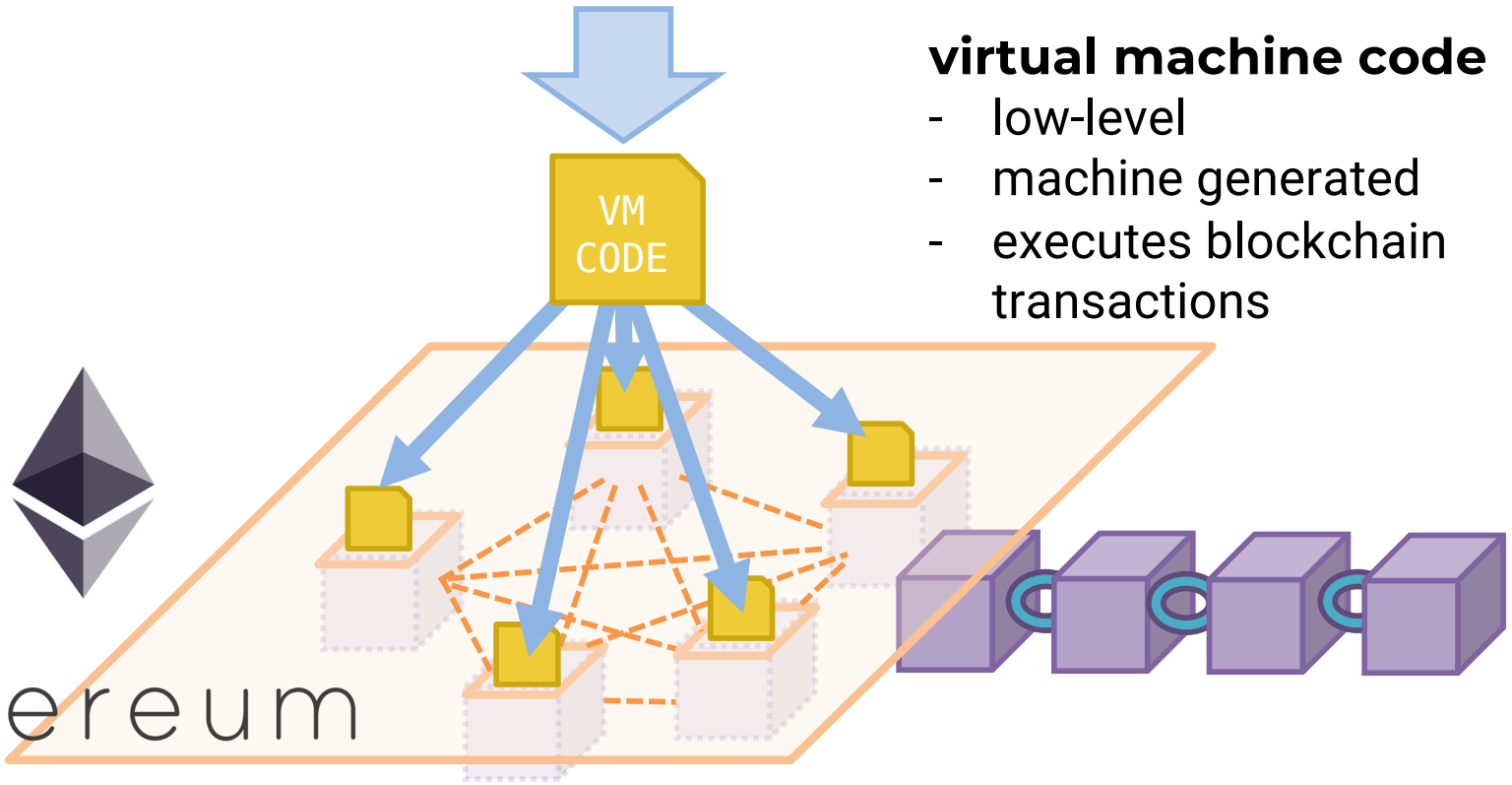
**contracts offer blockchain services**
that can be invoked by other smart contracts

# Smart Contracts



**smart contract**
- high-level
- human written
- programming language

**virtual machine code**
- low-level
- machine generated
- executes blockchain transactions

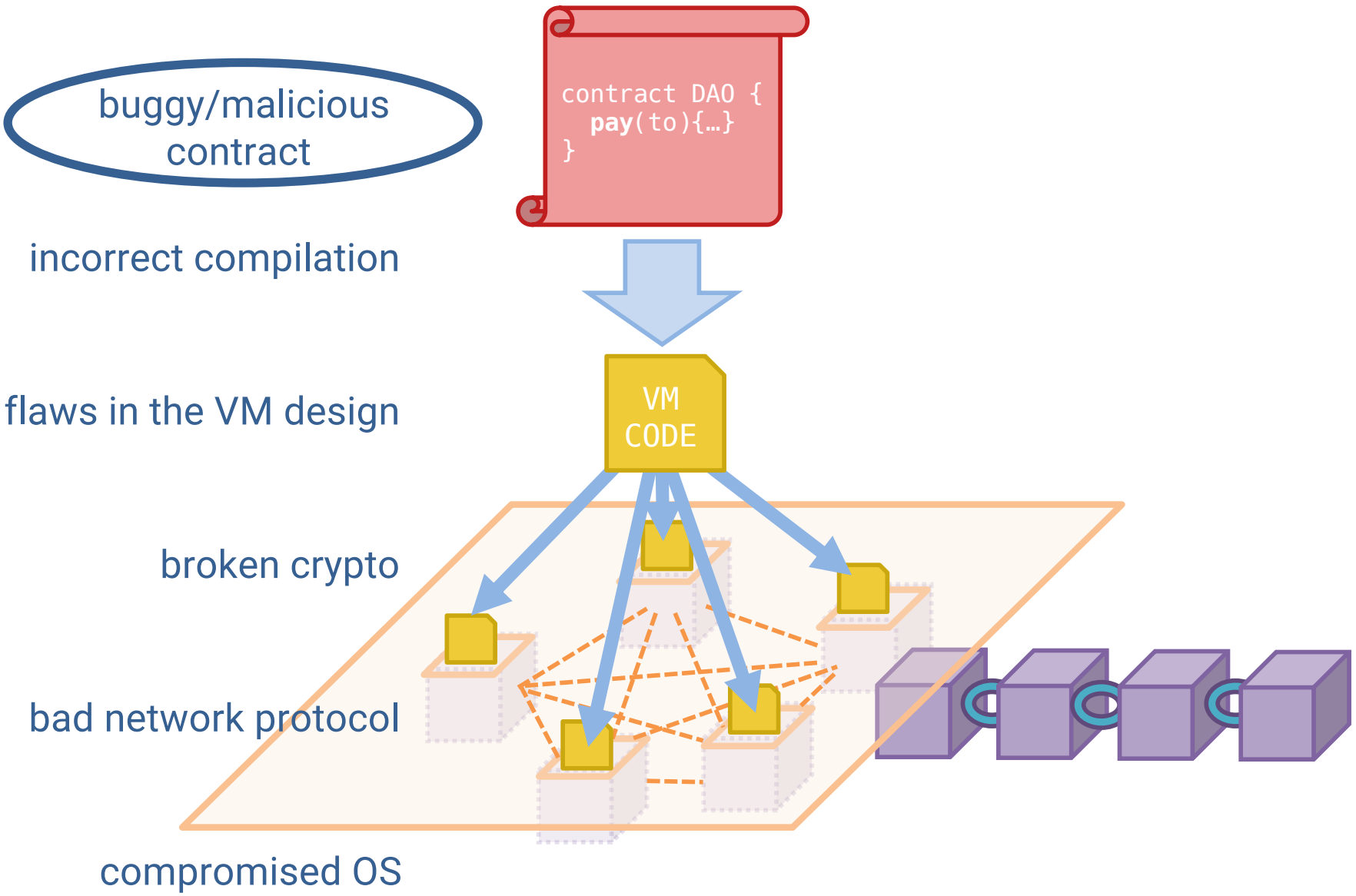





ethereum

**contracts offer blockchain services**
that can be invoked by other smart contracts

# Vulnerabilities?



# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

1. check the credit
2. transfer the amount
3. deduct the amount from available credit

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
       msg.sender.call.value(amount);
       credit[msg.sender] -= amount;
    }
  }

  function getCredit(address to) {…}
}
```

# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
      ms
      cr
    }
  }

  functio
}
```

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
       msg.sender.call.value(amount);
       credit[msg.sender] -= amount;
    }
  }

  function getCredit(add
}
```

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
       msg.sender.call.value(amount);
       credit[msg.sender] -= amount;
    }
  }

  function getCredit(add
}
```

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

after setup, invoke the fallback method (1)

# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
       msg.sender.call.value(amount);
       credit[msg.sender] -= amount;
    }
  }

  function getCredit(add
}
```

2 it calls **withdraw**, and the credit check succeeds

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
       msg.sender.call.value(amount);
       credit[msg.sender] -= amount;
    }
  }

  function getCredit(add
}
```

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

3 the call operation transfers some ether to Mallory, by calling the fallback method again!

# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
       msg.sender.call.value(amount);
       credit[msg.sender] -= amount;
    }
  }

  function getCredit(add
}
```

4 it calls **withdraw** again, and the credit check *still* succeeds!

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
       msg.sender.call.value(amount);
       credit[msg.sender] -= amount;
    }
  }

  function getCredit(add
}
```

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

5 the call operation transfers some ether to Mallory, by calling the fallback method again!

# Smart Contract Vulnerabilities

[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
       msg.sender.call.value(amount);
       credit[msg.sender] -= amount;
    }
  }

  function getCredit(add
}
```

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

6 And the loop continues until SimpleDAO runs out of ether, or the attack runs out of “gas”.

# Smart Contract Vulnerabilities

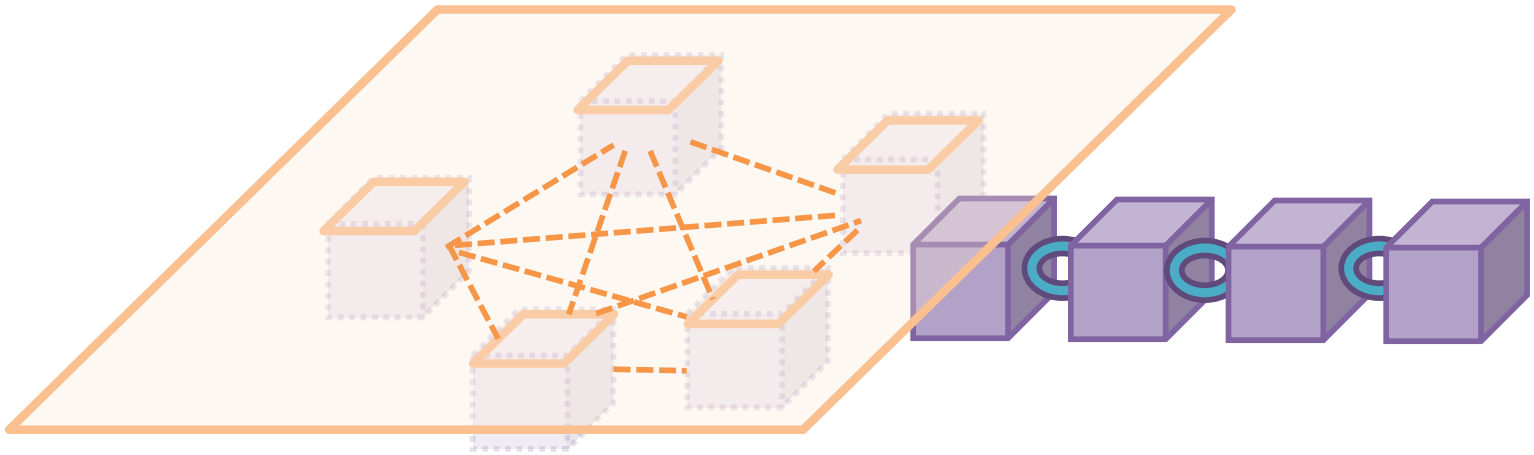
[Atzei, et al. 2017]

```
contract SimpleDAO {
  mapping (address ⇒ uint) public credit;

  function pay(address to) {
    credit[to] += msg.value
  }

  function withdraw(uint amount) {
    if (credit[msg.sender] >= amount) {
      credit[msg.sender] -= amount;
      msg.sender.call.value(amount);
    }
  }

  function getCredit(add
}
```

**simple fix:**
decrement the credit *before* transferring the amount.

```
contract Mallory {
  SimpleDao public dao = SimpleDao(0x354…);

  address owner;

  function Mallory(){owner = msg.sender;}

  function (){ dao.withdraw(dao.getCredit(this)); }

  function stealItAll() { owner.send(this.balance); }
}
```

# Formal Verification

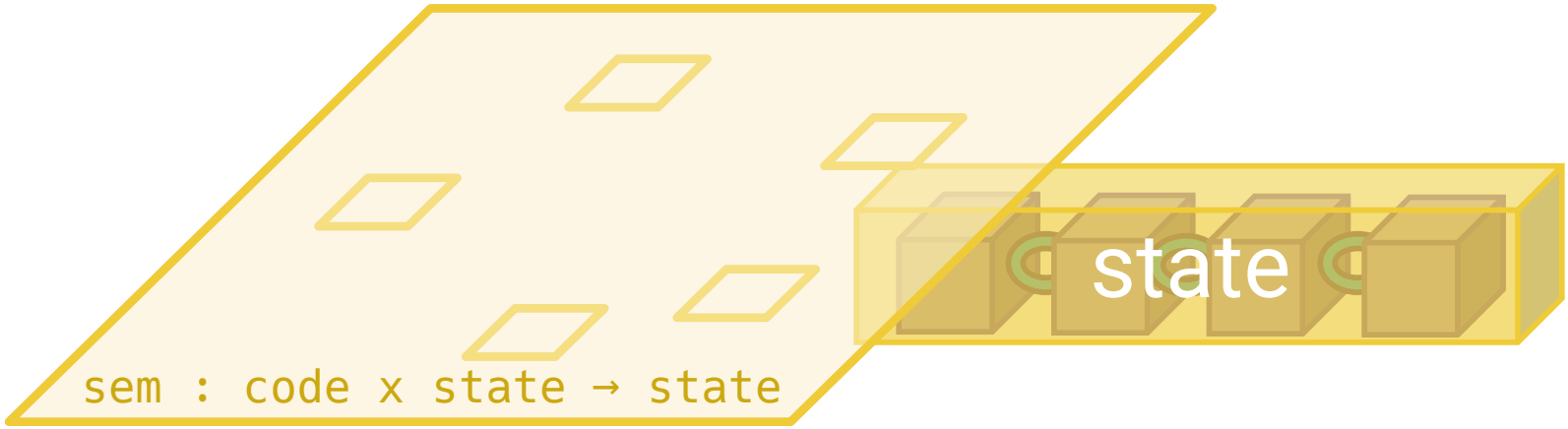


# Deep Specification

**mathematical model**

- hides implementation details
- describes the system behaviors
- implemented in an *interactive theorem prover*

1. Coq
2. LEAN
3. F*
4. Isabelle/HOL
5. Agda



# Correctness Definition

[Grischenko, et al. 2018]

For the DAO attack, the problem can be characterized as a failure of a property known as **call integrity,** reentrant code controlled by attacker.

```
Definition call_integrity (p:code) :=
   ∀(s1, s2:state),
    (sem (p,s1) = s1') →
    (sem (p,s2) = s2' → c(s1') ~ c(s2').
```

Define a predicate that ood:

```
Definition OK (p:

Theorem call_safet
  call_integrity ⟦
```

**Specification:**
An attacker shouldn't be able

**Conformance check:**
Says how to determine whether a program satisfies the specification.

# Formal Verification

1. Embed the smart contract
2. Use the formal semantics to check its properties


This … hides a lot of (sometimes) manual effort and a very large proof.

```
⟦DAO contract⟧ : code = {…}

Lemma buggy_DAO : not (OK(DAO)).
Proof. …
Qed.

Lemma safe_FixedDAO :
Proof. …
Qed.
```

**⇒ proof engineering**

- constructing proofs at scale
- modeling techniques
- proof principles
- tactics, automation

# Formal Verification ⇒ Fewer Vulnerbilities

malicious contract

incorrect compilation

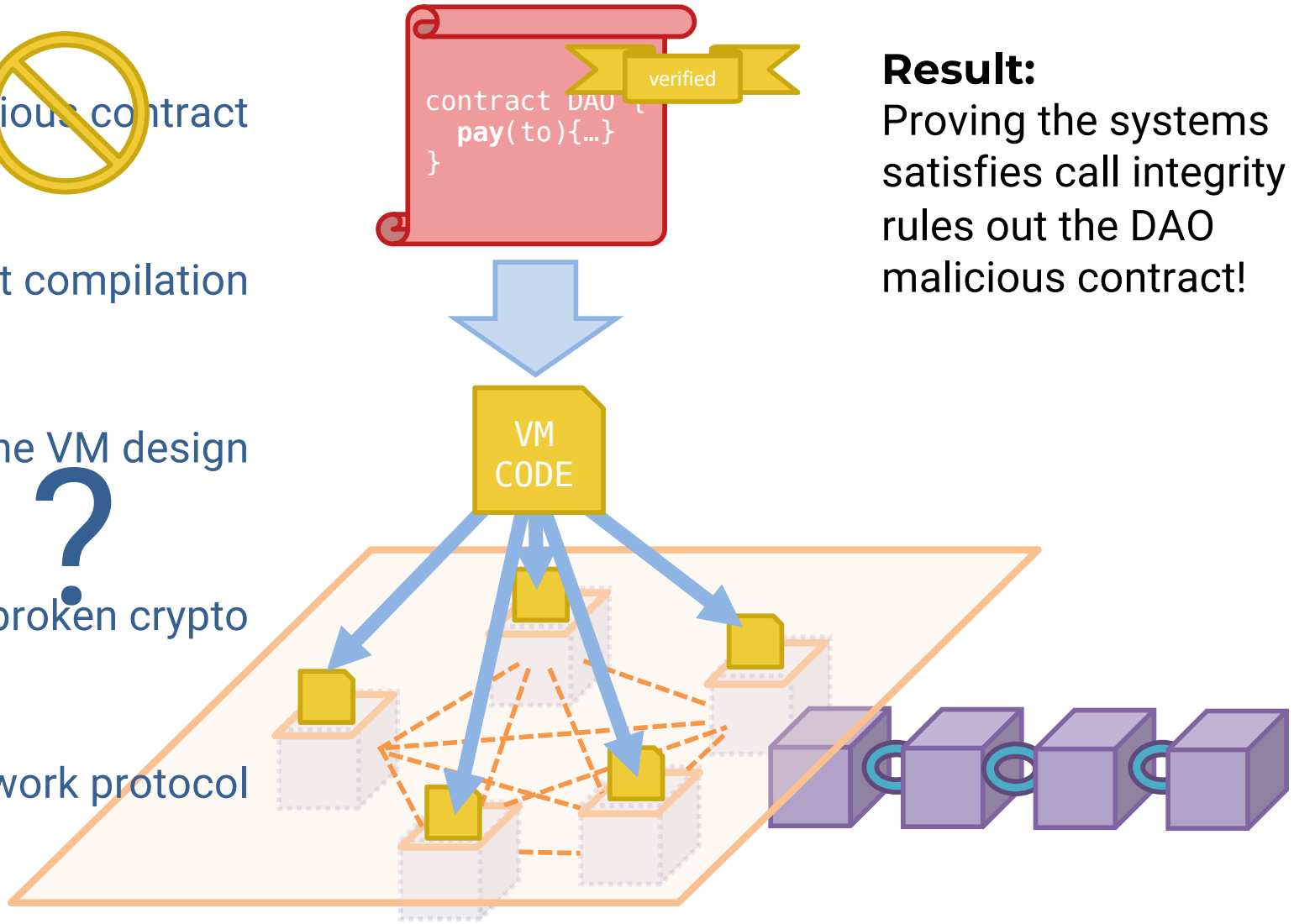
flaws in the VM design

?

broken crypto

bad network protocol

compromised OS




**Result:**
Proving the systems satisfies call integrity rules out the DAO malicious contract!

# VELLVM – VERIFIED LLVM IR

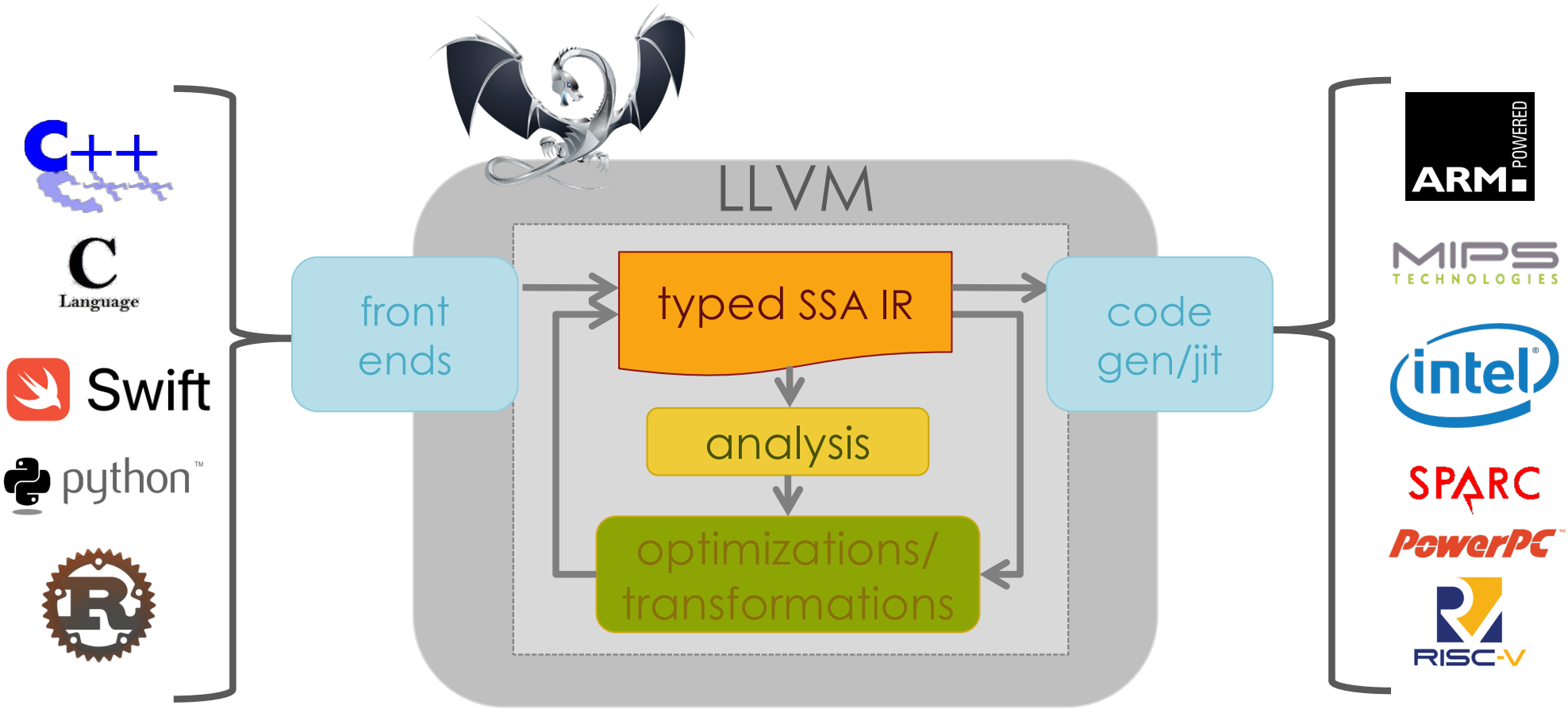
1. LLVM IR
   - semantics subtleties: `undef`
2. Implementing LLVM IR Semantics in Coq
   - interaction trees & layered interpreters
   - nondeterminism
3. Applications of Vellvm
   - correctness proofs
4. Conclusions

# VELLVM – VERIFIED LLVM IR

1. **LLVM IR**
   - semantics subtleties: `undef`
2. Implementing LLVM IR Semantics in Coq
   - interaction trees & layered interpreters
   - nondeterminism
3. Applications of Vellvm
   - correctness proofs
4. Conclusions

# LLVM Compiler Infrastructure

[Lattner,2002]



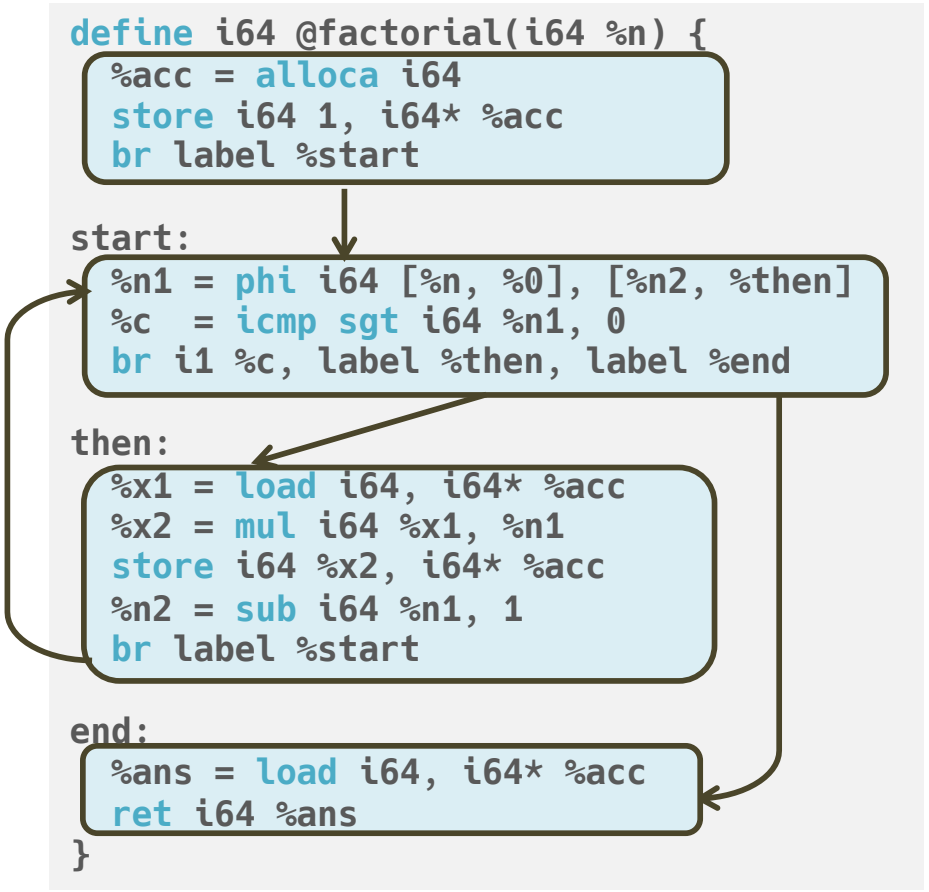
# Translating C to LLVM IR

```
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
}
```

# Translating C to LLVM IR

```
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
}
```




# LLVM IR

=

Control-flow Graphs:

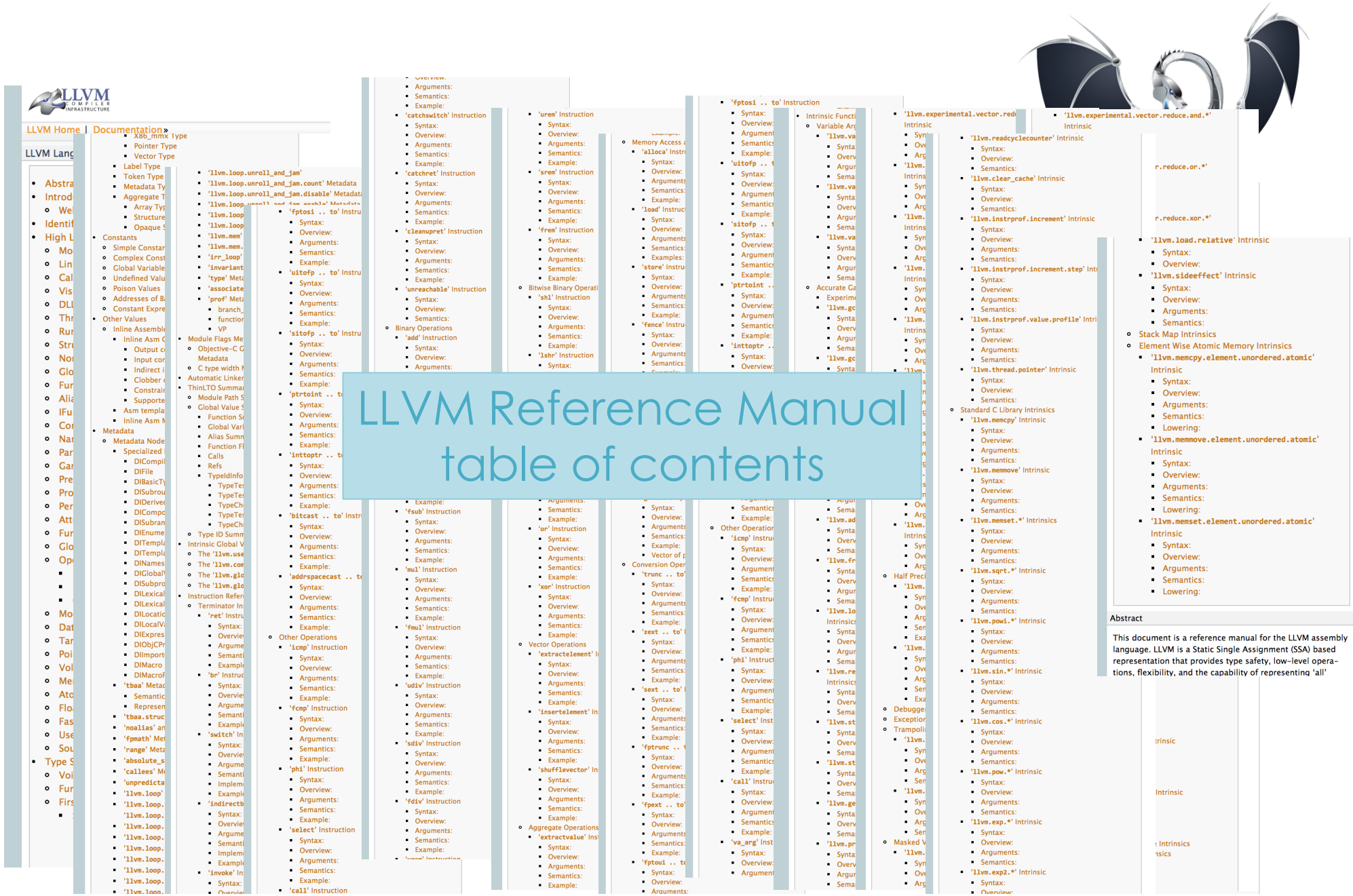
- Labeled blocks
- Straight-line Code
  - simple instructions
- Block Terminators
  - (conditional) jumps
- Static Single Assignment

# Other LLVM IR Features

- **C-style data values**
  - ints, structs, arrays, pointers, vectors
- **Memory model & type system**
  - used for layout/alignment/padding
- **Unsafe primitives**
  - ptrtoint and inttoptr casts
- **Undefined behavior**
  - poison values
- **Intrinsics**
  - additional instructions

Make targeting LLVM IR easy and attractive for developers!

# But… it's complex



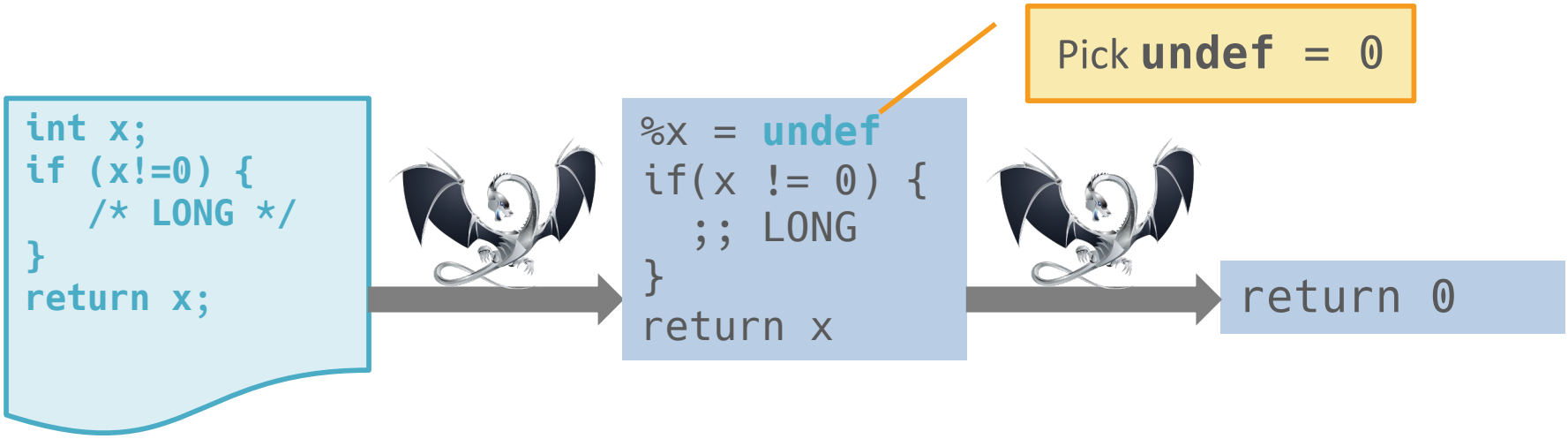
# One Example: `undef`

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

# Optimization with undef




"Optimistically" use the best value
⇒ significantly faster code!

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:

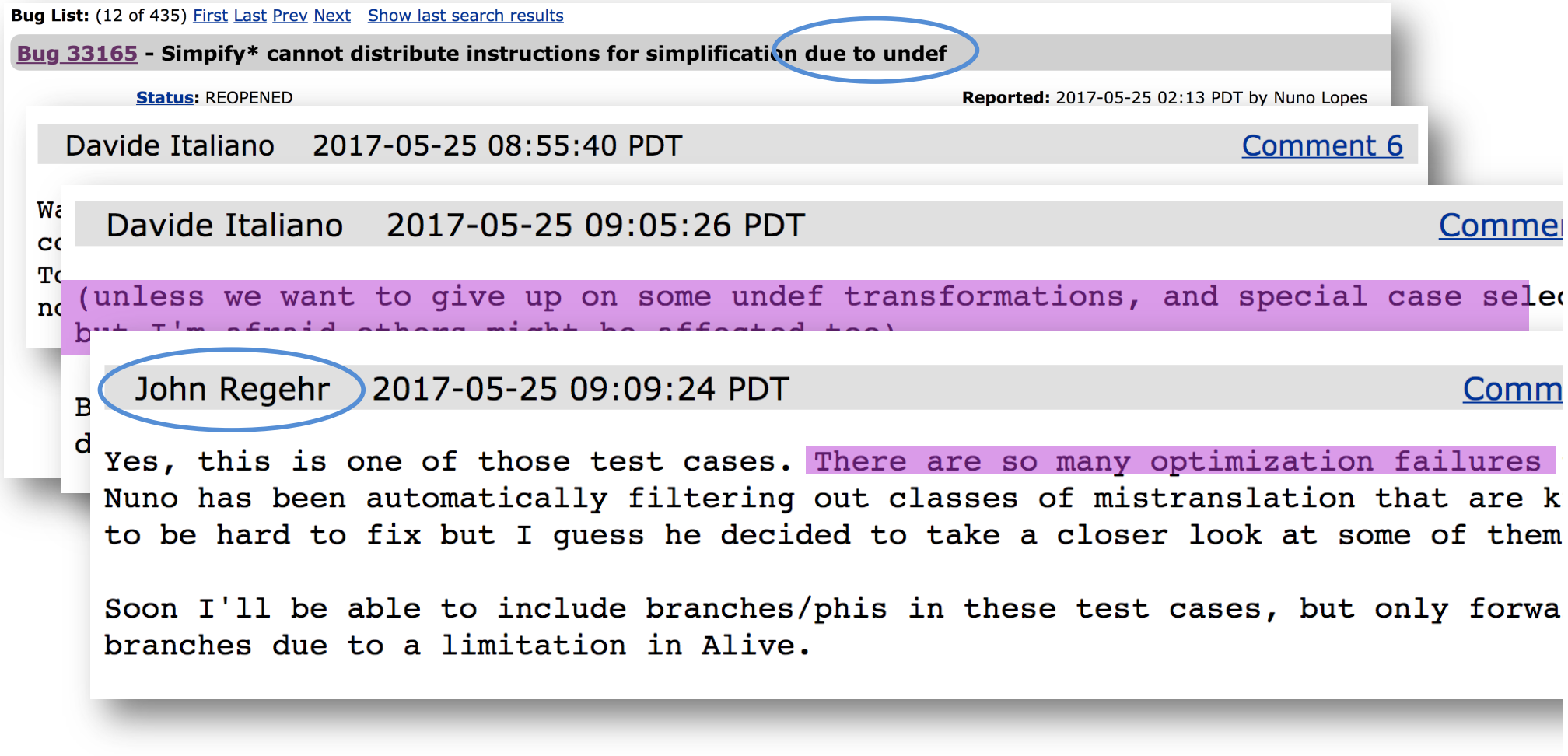
```
%y = mul i8 %x, 2
```

versus:

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

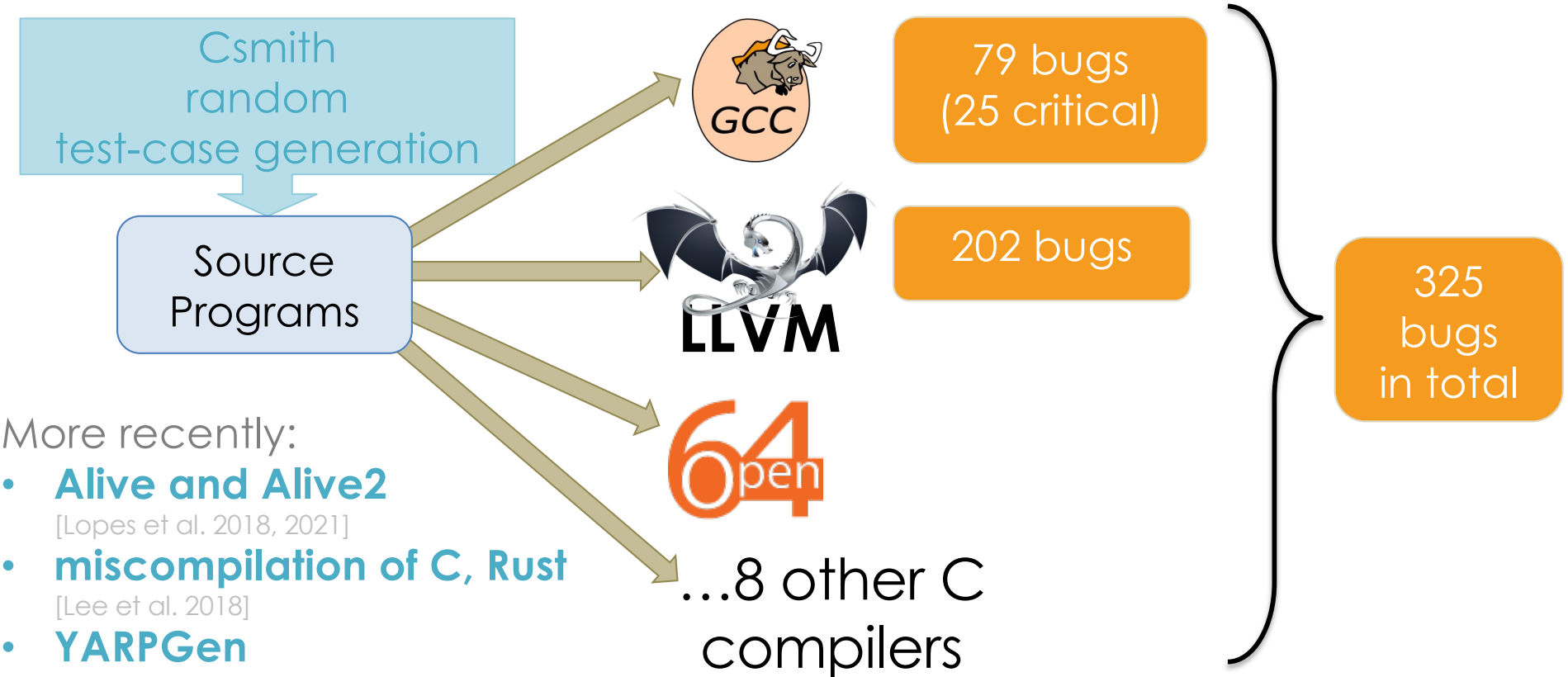
Upshot: if **%x** is **undef**, we can't optimize **mul** to **add**

# What's the problem?



# Compiler Bugs

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




More recently:

- **Alive and Alive2**
  [Lopes et al. 2018, 2021]
- **miscompilation of C, Rust**
  [Lee et al. 2018]
- **YARPGen**
  [Livinski et al. 2020]

LLVM might be hard to trust
(especially for critical code)

What can we do about it?

# Our Approach: Formal Verification

***Interactive theorem proving***

- Coq (or Lean )
- human-in-the-loop
- (not model checking / SMT)

Using Coq *is* programming

…but some of your programs *are* proofs

[coq.inria.fr]

LEAN

⇒ *proof* engineering

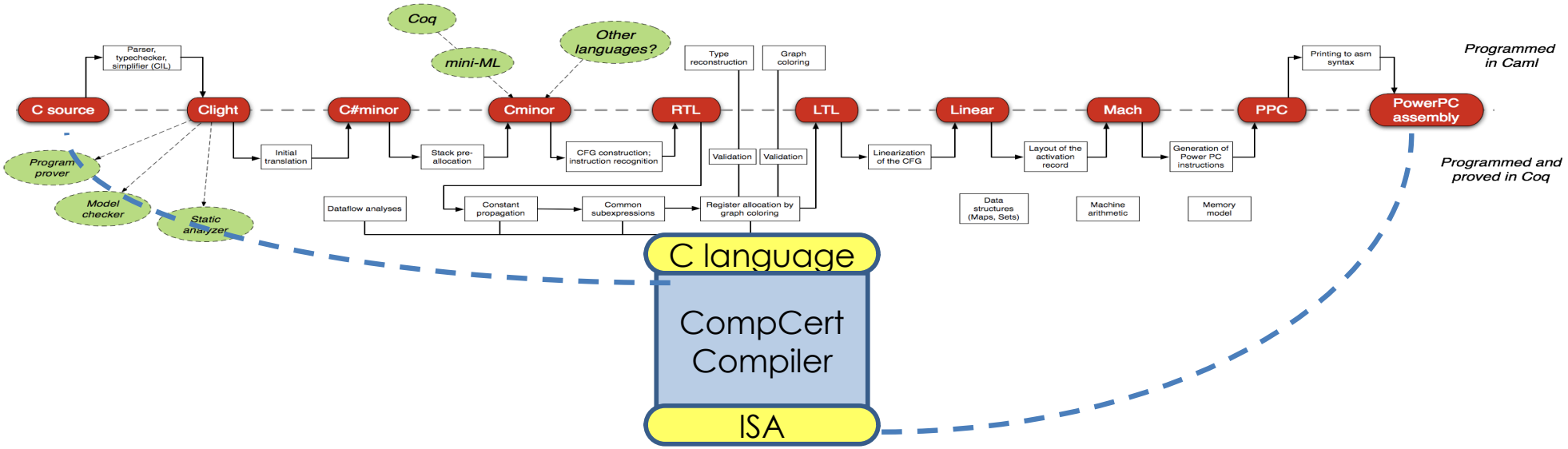
# Inspiration: CompCert

[Leroy and Blazy et al. INRIA ]

Optimizing C Compiler:

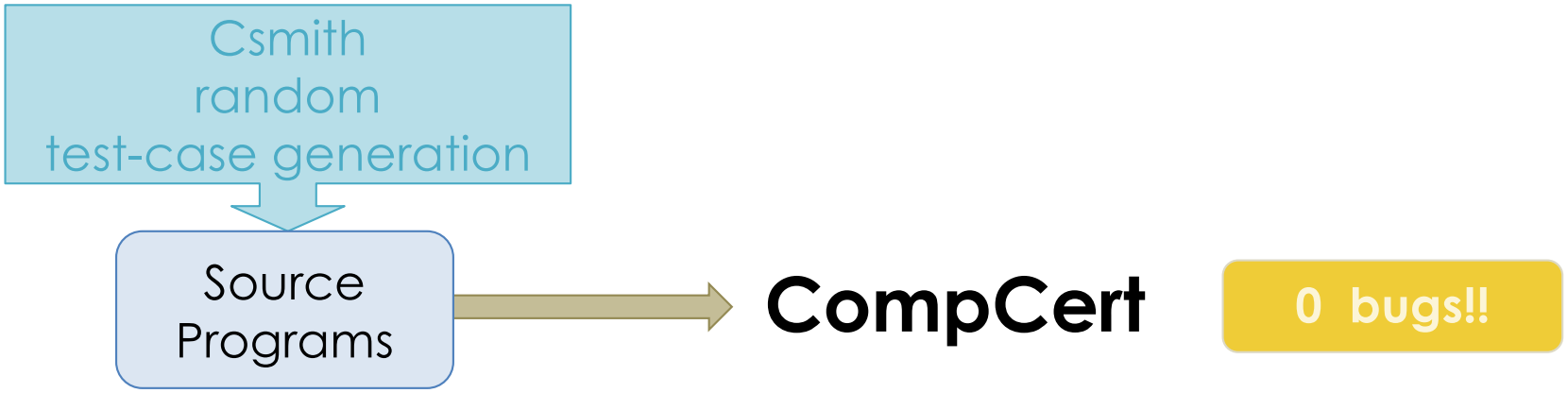
proved correct end-to-end

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.***"

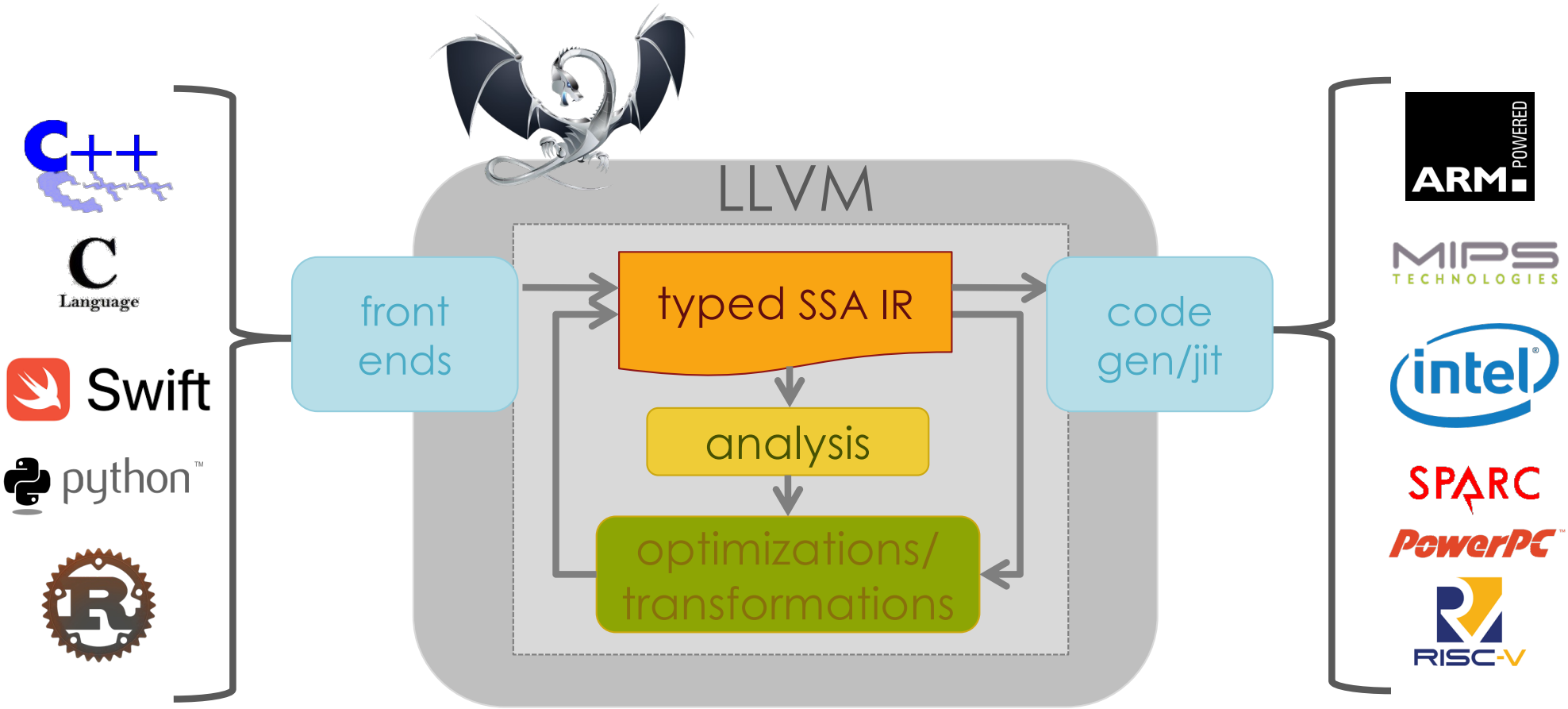
*– Regehr et al., PLDI 2011*

# VELLVM – VERIFIED LLVM IR

1. LLVM IR
   - semantics subtleties: `undef`
2. **Implementing LLVM IR Semantics in Coq**
   - interaction trees & layered interpreters
   - nondeterminism
3. Applications of Vellvm
   - correctness proofs
4. Conclusions

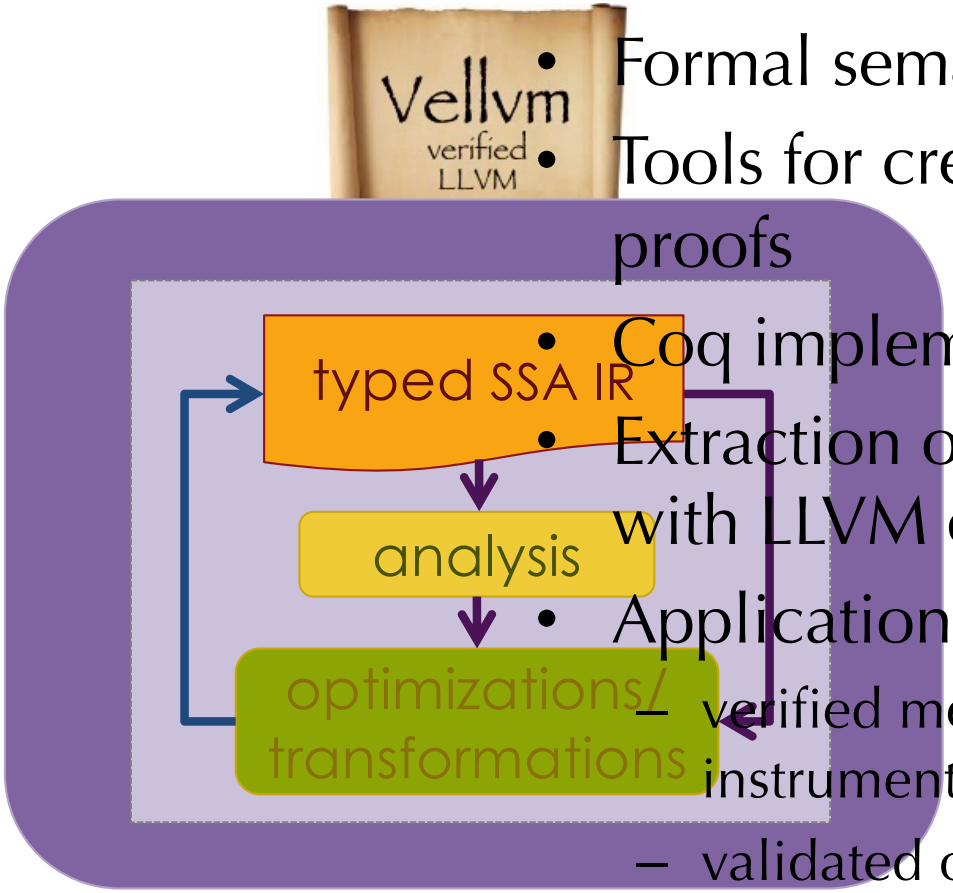
# LLVM Compiler Infrastructure

[Lattner,2002]



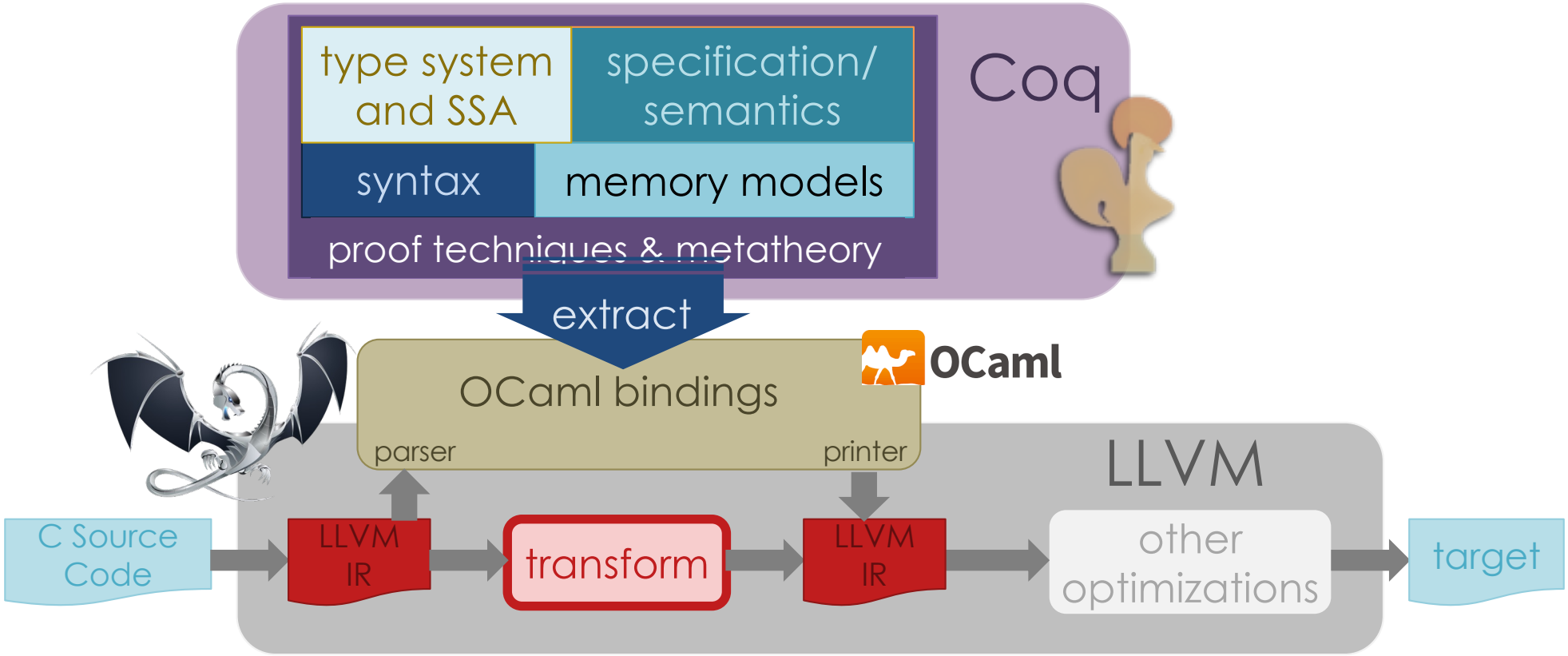
# The Vellvm Project

[POPL'12, CPP'12, PLDI'13, CAV'15,ICFP'20,ongoing…]

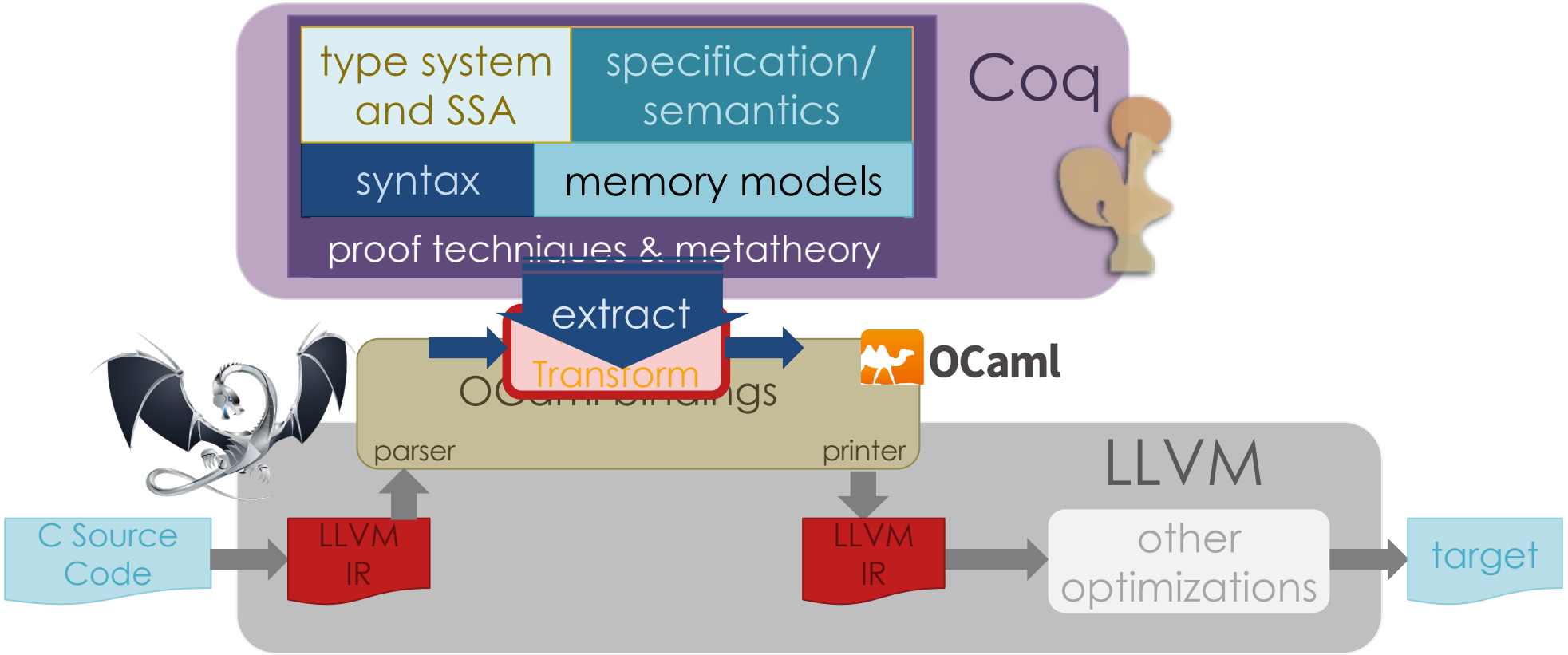



- Formal semantics
- Tools for creating correctness proofs
- Coq implementation
- Extraction of passes for use with LLVM compiler
- Applications:
  - verified memory safety instrumentation
  - validated optimizations

# Vellvm Framework



# Vellvm Framework



How do we define a formal specification of LLVM IR semantics?

# LLVM IR Semantics

SSA ≈ functional program [Kelsey 1995 / Appel 1998]

+

- **`undef` values**
- **Side Effects**
  - structured heap loads/stores
  - system calls (I/O)
  - undefined behavior
- **Types & Memory Layout**
  - structured, recursive types
  - stack allocation
  - ptrtoint type casts

# Writing Interpreters in Coq

**Gallina (Coq's Language)**

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

```
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

How do we write the interpretation function?

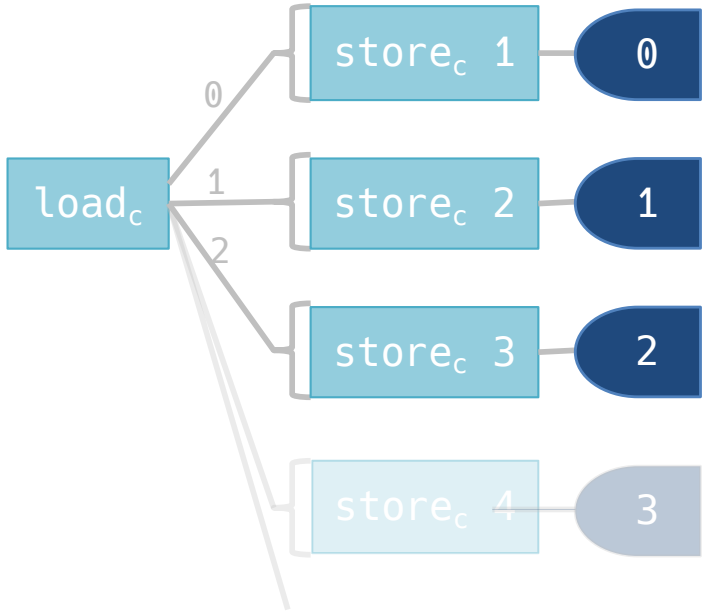
# Interaction Trees

[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



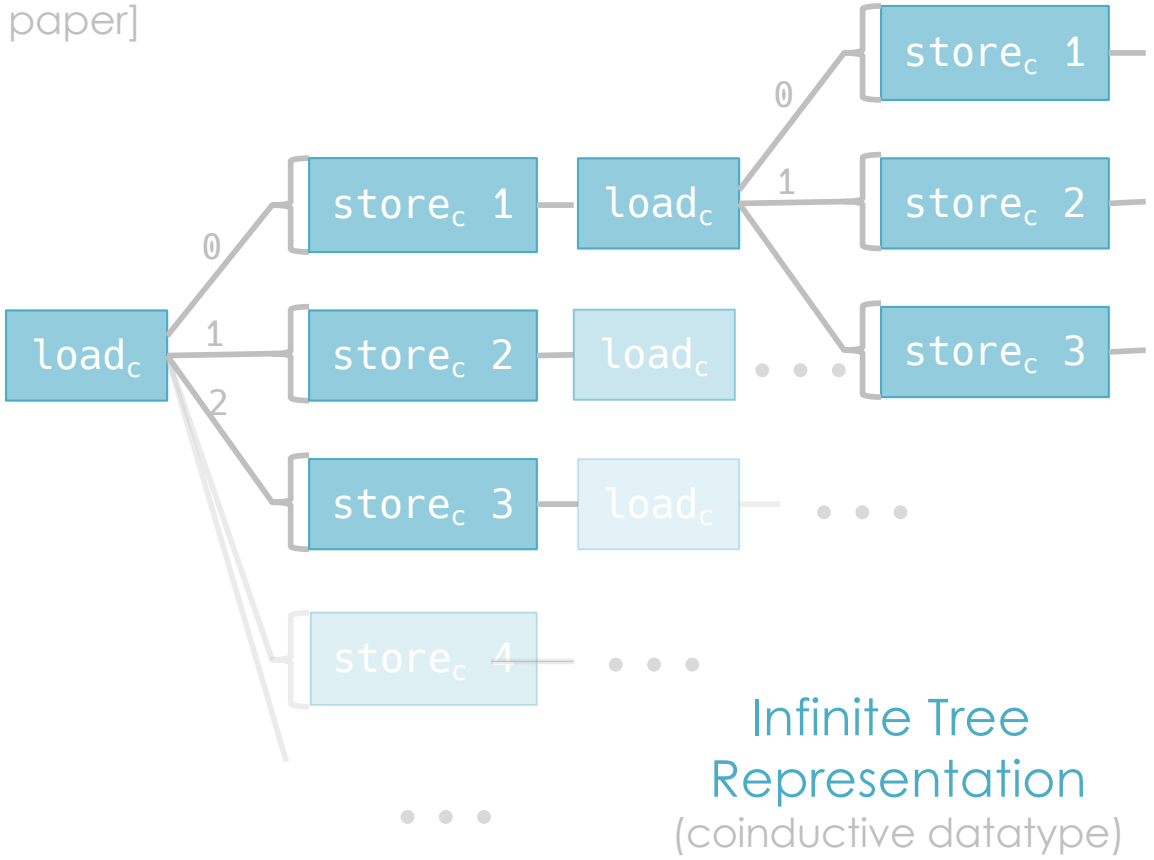

Interaction Tree Representation

# 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)

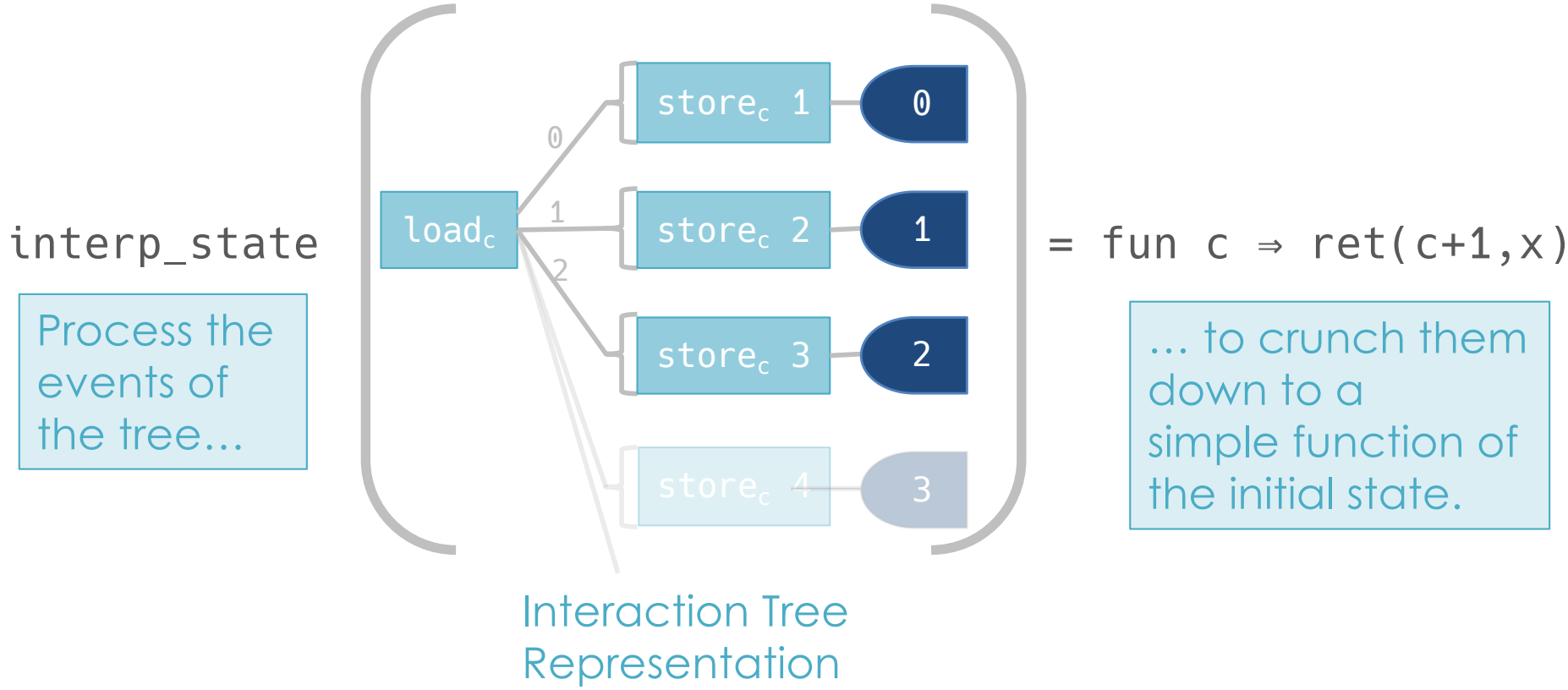
# Good Qualities of Interaction Trees

- `ITree E` is a **monad**
  - models sequential computation
  - **`bind`** is grafts on subtrees
- **Extractable** from Coq into OCaml/Haskell

  ⇒ we can run the computations
- **Behavioral Equivalences**
  - *strong bisimulation*
  - *weak bisimulation*
    (insert finite # taus before any event)
  - rich equational theory

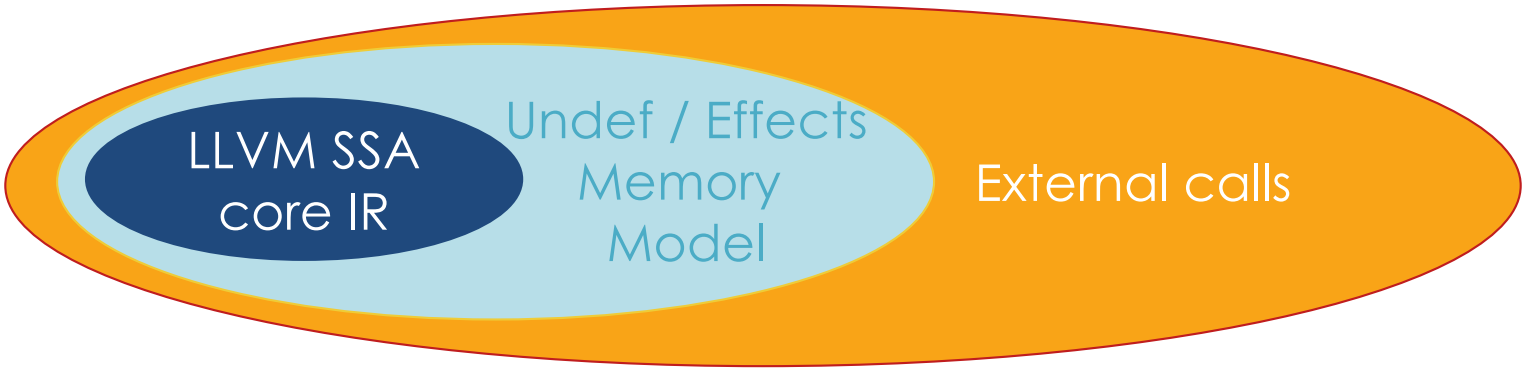
Quite intricate coinductive proofs needed here…
… but, they're encapsulated in the library. [CPP 2020]

# Interpreters over ITrees



# Defining Modular Semantics

- Core: LLVM control-flow-graph interpreter
- `undef` values / computation
- Memory Model [CAV 15,PLDI 15, ????24]
  - interprets loads/stores
  - nondeterminism for allocation
  - support for casts

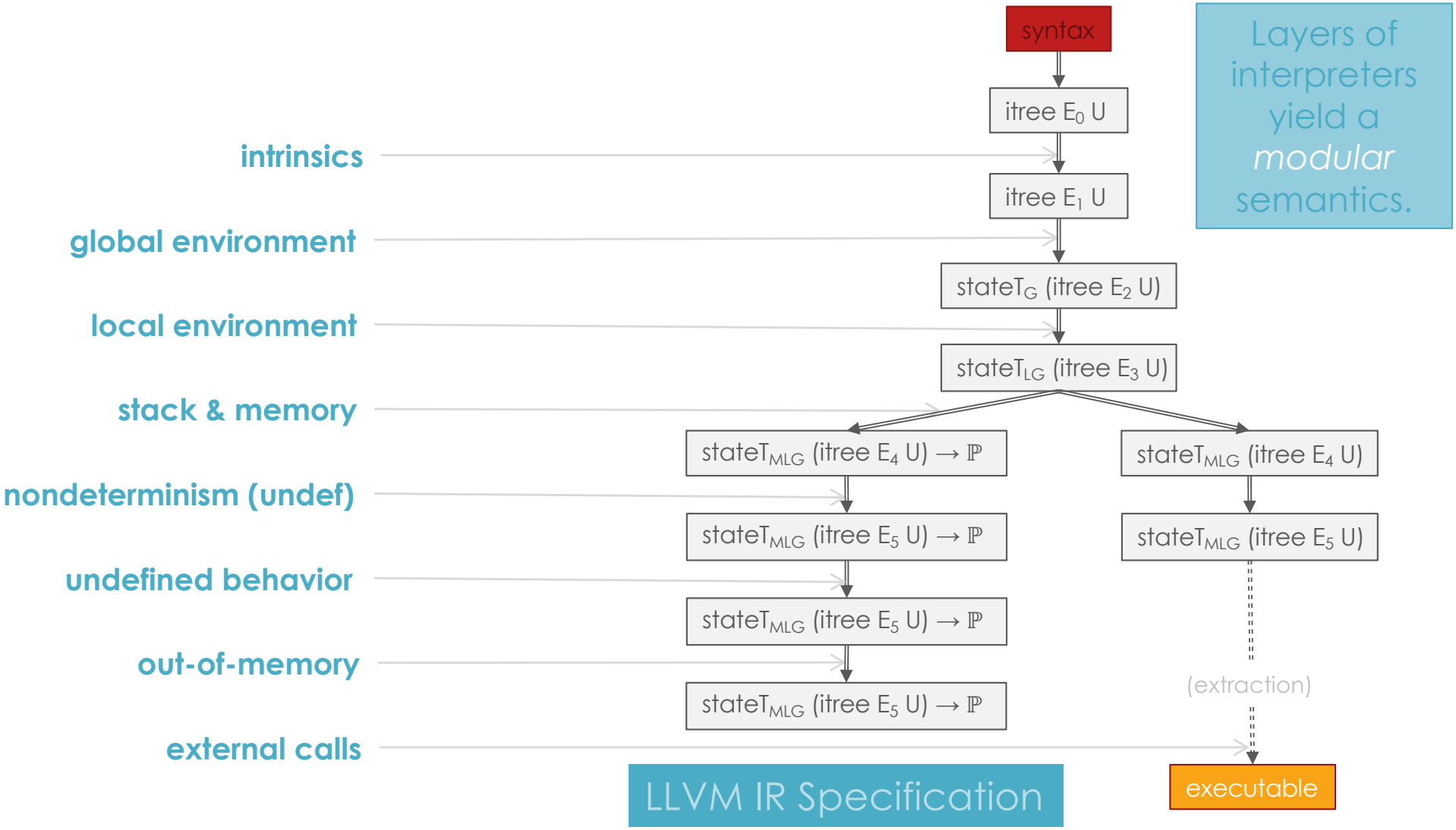


# LLMV Memory Model Events (simplified )

```
(* 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 – VERIFIED LLVM IR

1. LLVM IR
   - semantics subtleties: `undef`
2. Implementing LLVM IR Semantics in Coq
   - interaction trees & layered interpreters
   - nondeterminism
3. **Applications of Vellvm**
   - correctness proofs
4. Conclusions

# 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

# Proving Optimizations:

- **Refinement of uvalues**

- **Behavioral refinement of computations**

$$[\![\texttt{i8 undef}]\!] = \{0,\ldots,255\}$$
$$[\![\texttt{i8 1}]\!] = \{1\}$$
$$\text{so:} \quad [\![\texttt{i8 undef}]\!] \supseteq [\![\texttt{i8 1}]\!]$$

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

$$[\![P]\!]m \supseteq [\![Q]\!]m$$

Q is a valid optimization of P.

# Proofs *are* programs!

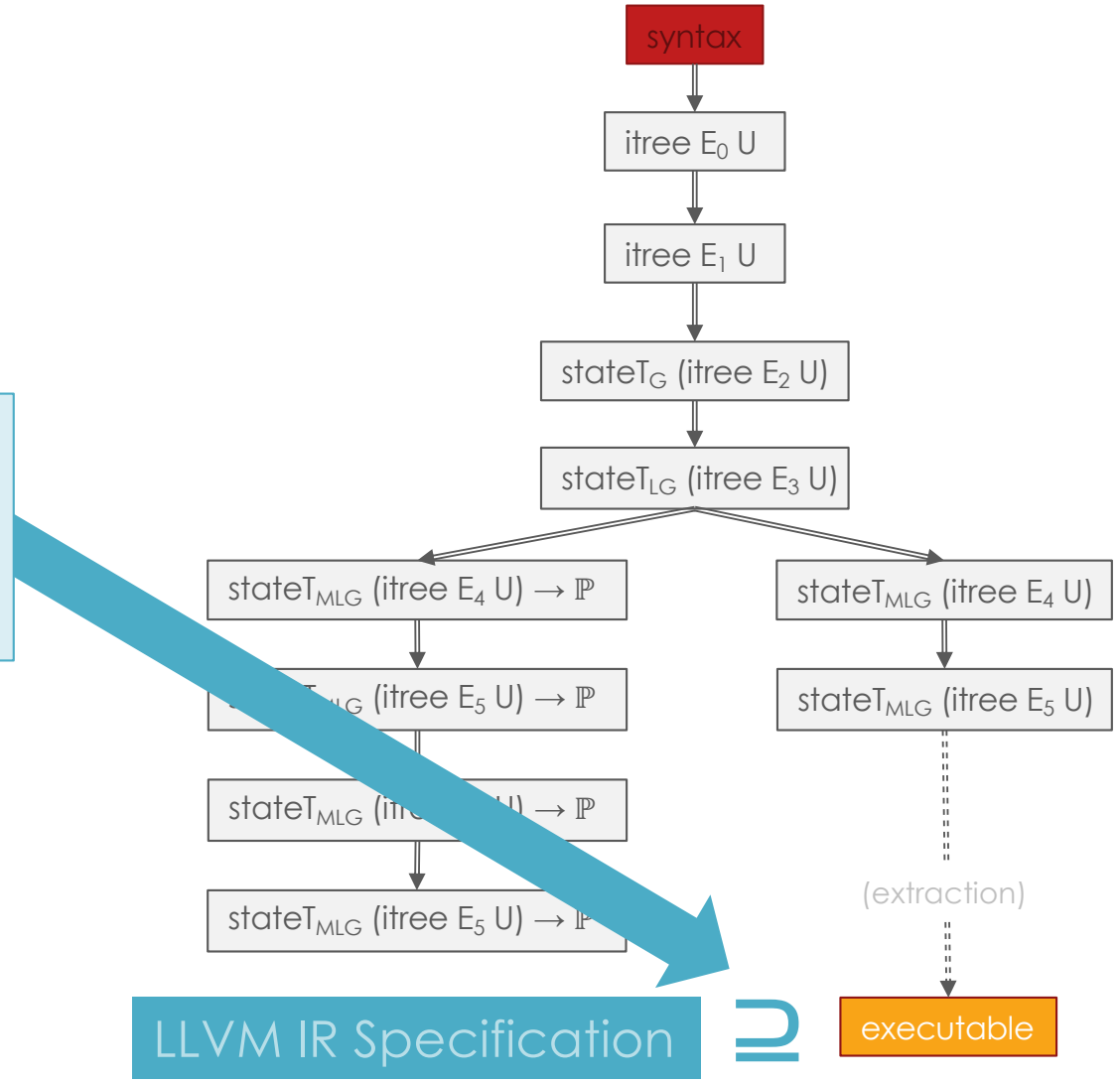
```
Theorem mul_add_correct :
  ∀ x, x ≠ undef → ⟦mul i64 2, x⟧ ⊒ ⟦add i64 x, x⟧.
Proof.
  …
Qed.
```

**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 specificiation.*



# 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!

# Reasoning about LLVM IR code

- Velliris: Vellvm + Iris Separation logic
  - Verified reasoning about behaviors of LLVM IR code
  - Example: Loop-invariant Code Motion

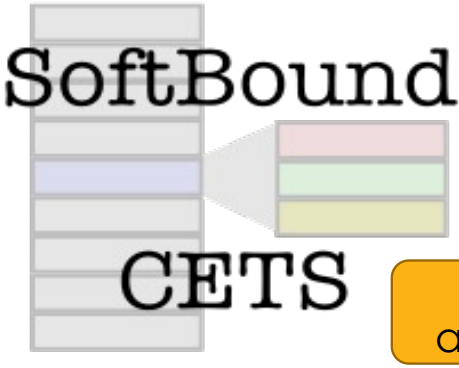
[Irene Yoon 2023]

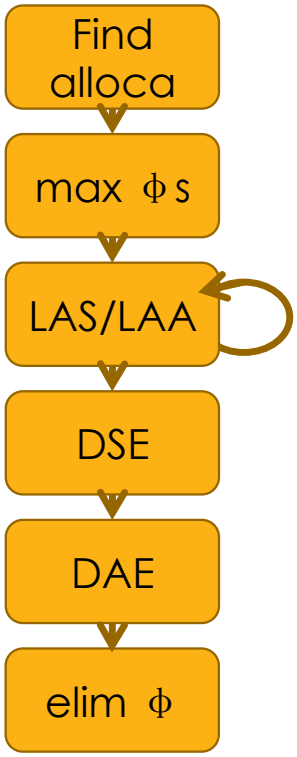
```
void increment(int *n);
int get_int(int *x) {
  int *n; int i = 0; n = &i;
  while (*n < *x)
    { increment (n); }
  return *n;
}
```

```
void increment(int *n);
int get_int(int *x) {
  int *n; int i = 0; n = &i;
  int y = *x;
  while (*n < y)
    { increment (n); }
  return *n;
}
```

# 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]

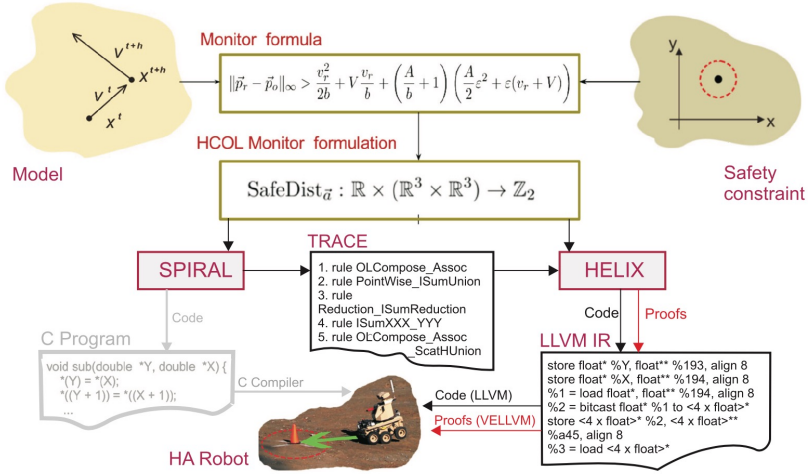


# Spiral/Helix

[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 Coq, targets Vellvm
- Bottom of the compilation chain proved




Exectuability

# VELLVM DEMO

# So What?

- Debugging
- Validate Vellvm Semantics against other implementations
  - test suite of ~140 "semantic tests"
  - e.g., integrate with QuickChick, Csmith, ALIVE
- 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 about the LLVM compiler

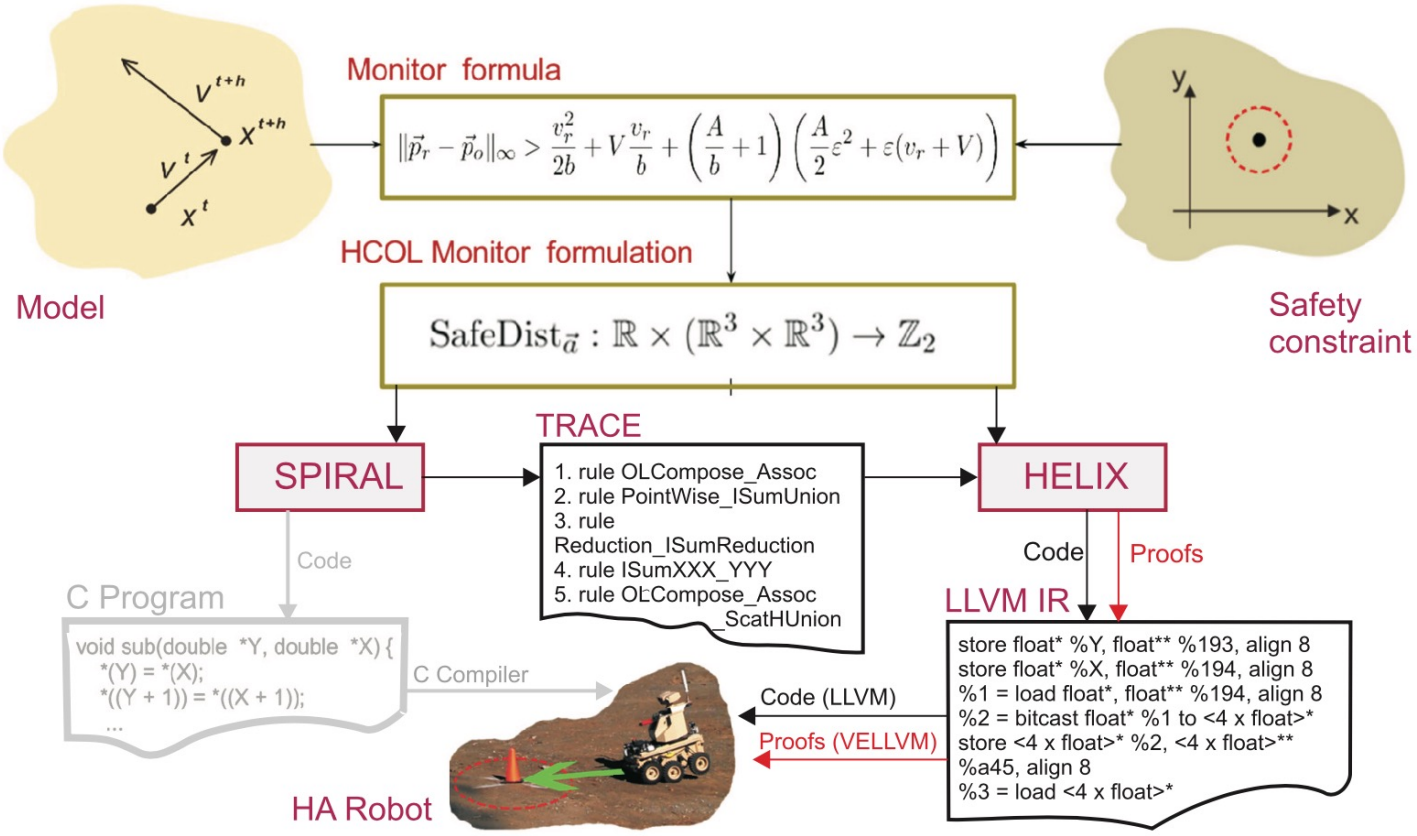
# Deep Specifications


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

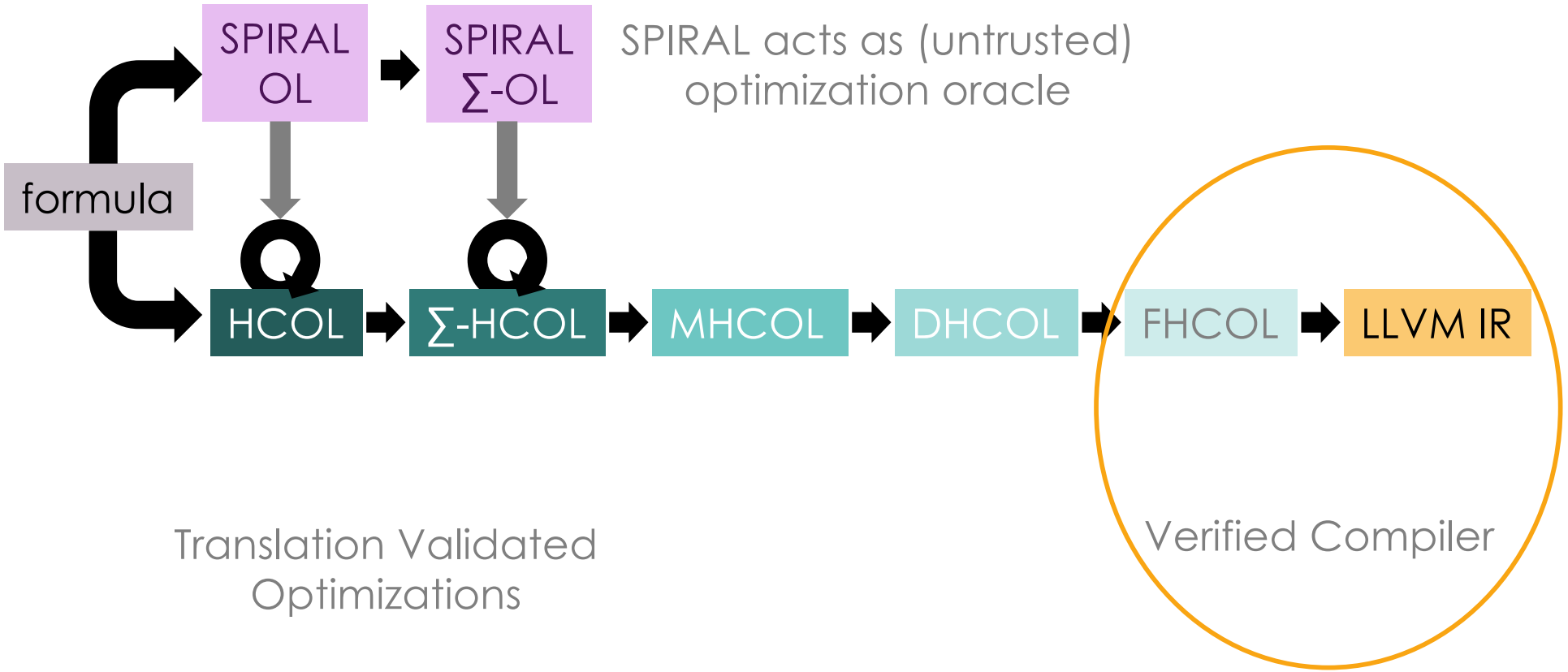
# SPIRAL / HELIX

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

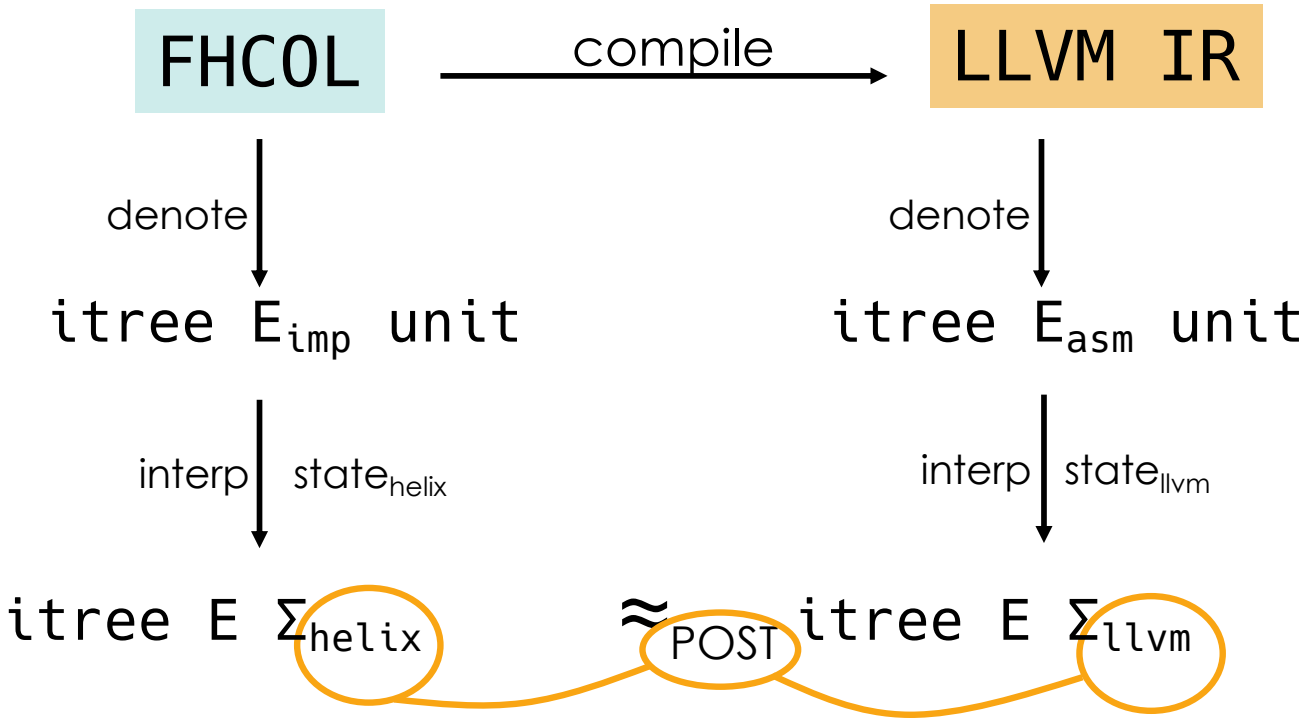
DSL for high-performance numerical computing.



# HELIX Compilation Pipeline



# Compiler Correctness Proof



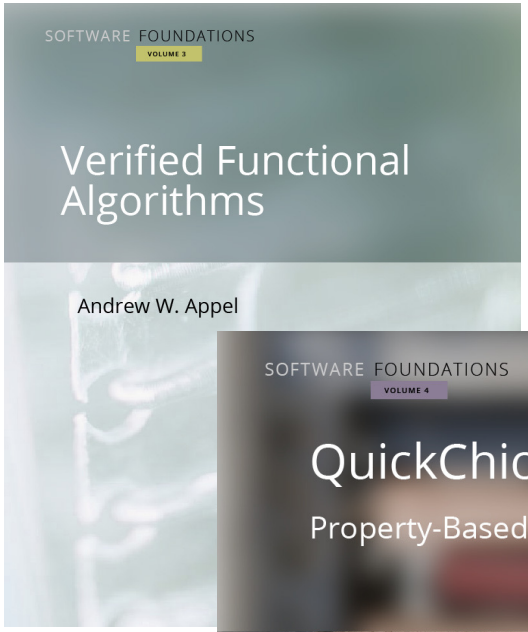
# WHAT NEXT?

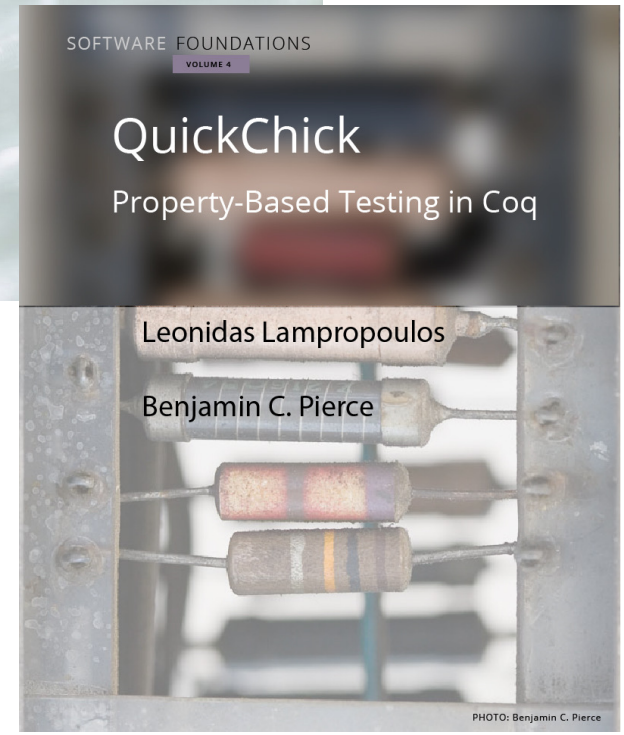
# After CIS 5000

- **Courses at Penn**
  - CIS 5520 – Advanced Programming
  - CIS 4521/5521 - Compilers
- **Research Conferences:**
  - Certified Programs and Proofs (CPP)
  - Interactive Theorem Proving (ITP)
  - Theorem Proving and Higher-order Logics (TPHOLS)
  - Principles of Programming Languages (POPL)
  - Programming Languages Design and Implementation (PLDI)
  - International Conference on Functional Programming (ICFP)
  - …
- **Other Theorem Provers:**
  - LEAN
  - Isabelle/HOL
  - Agda
  - F*

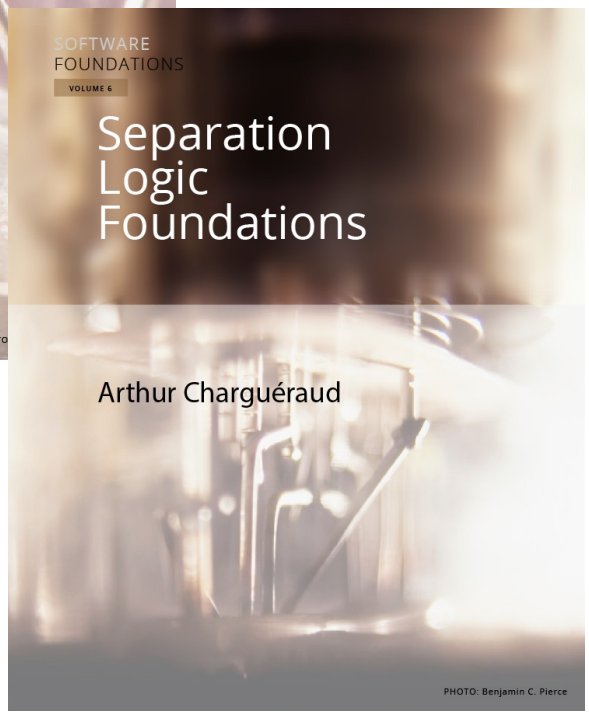
# After CIS 5000

- **More Software Foundations Volumes**
  - written in the same style as SF and PLF







# Thanks!

- To our three fantastic TAS:
  - Zain Aamer
  - Cuthbert Li
  - Elan Roth

And, thanks to all of you!