Wrap-up Lecture

CIS 5000: SOFTWARE FOUNDATIONS

Steve Zdancewic

Fall 2022

Announcements

- HW12: Sub.v
  - due: Monday, Dec. 12th at ***midnight***
  - ***NOTE:* See submission instructions on ED about the "short answer" questions.**

- Final exam:
  Tuesday, Dec. 20th 9:00 – 11:00AM
  Location: **Skirkanich Auditorium**

- Course status:
  - We will be working on compiling the course grades, statistics, etc., during the next week.
  - If you have questions about how you stand in the course, please contact me.

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.

Standard vs Advanced [WPE I]

- As with the previous exams there will be a “standard” track and an “advanced” track.

- PhD Students and WPE-I exam students must take the advanced track
  - It will include some form of written proof
  - WPE-I exams often include more “synthesis” problems (i.e., explore a variant language, proofs, etc.)

- See the example final exams that are available on the course web page.

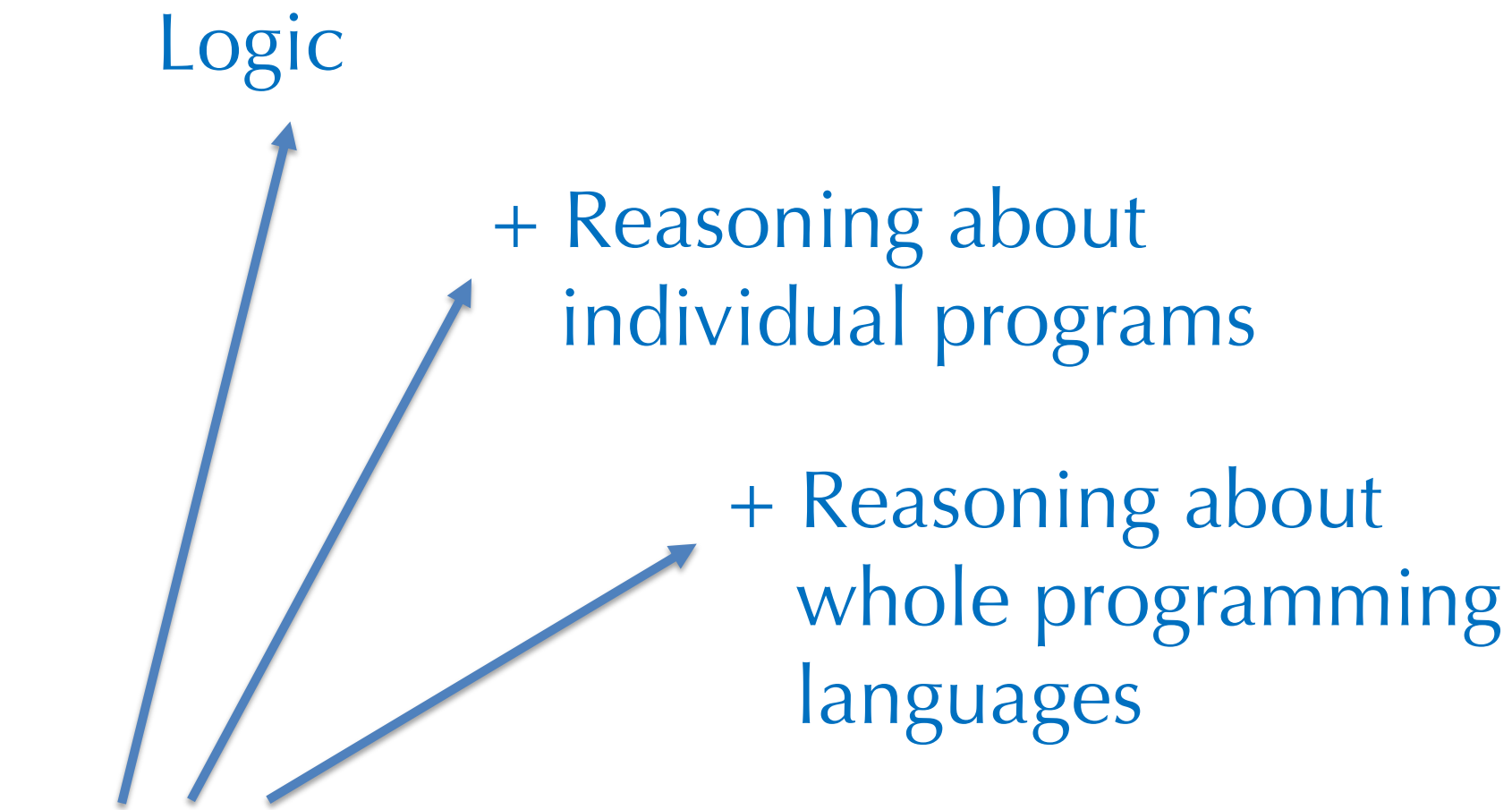
Remote Exam Option

- Some students with extenuating circumstances have asked to take the exam remotely.

- The remote option will be the same format.
- Released on Gradescope for a 24-hour duration starting at the same time as the in-person exam.
- There will be a 2-hour (+ a bit extra) window to download, complete, (scan if necessary), and upload the exam to Gradescope.

- **Send mail to me before Thursday, December 15th if you wish to be considered for the remote option.**

COURSE RECAP



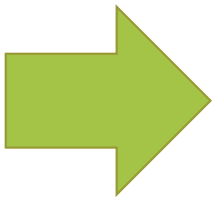
SOFTWARE FOUNDATIONS

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!

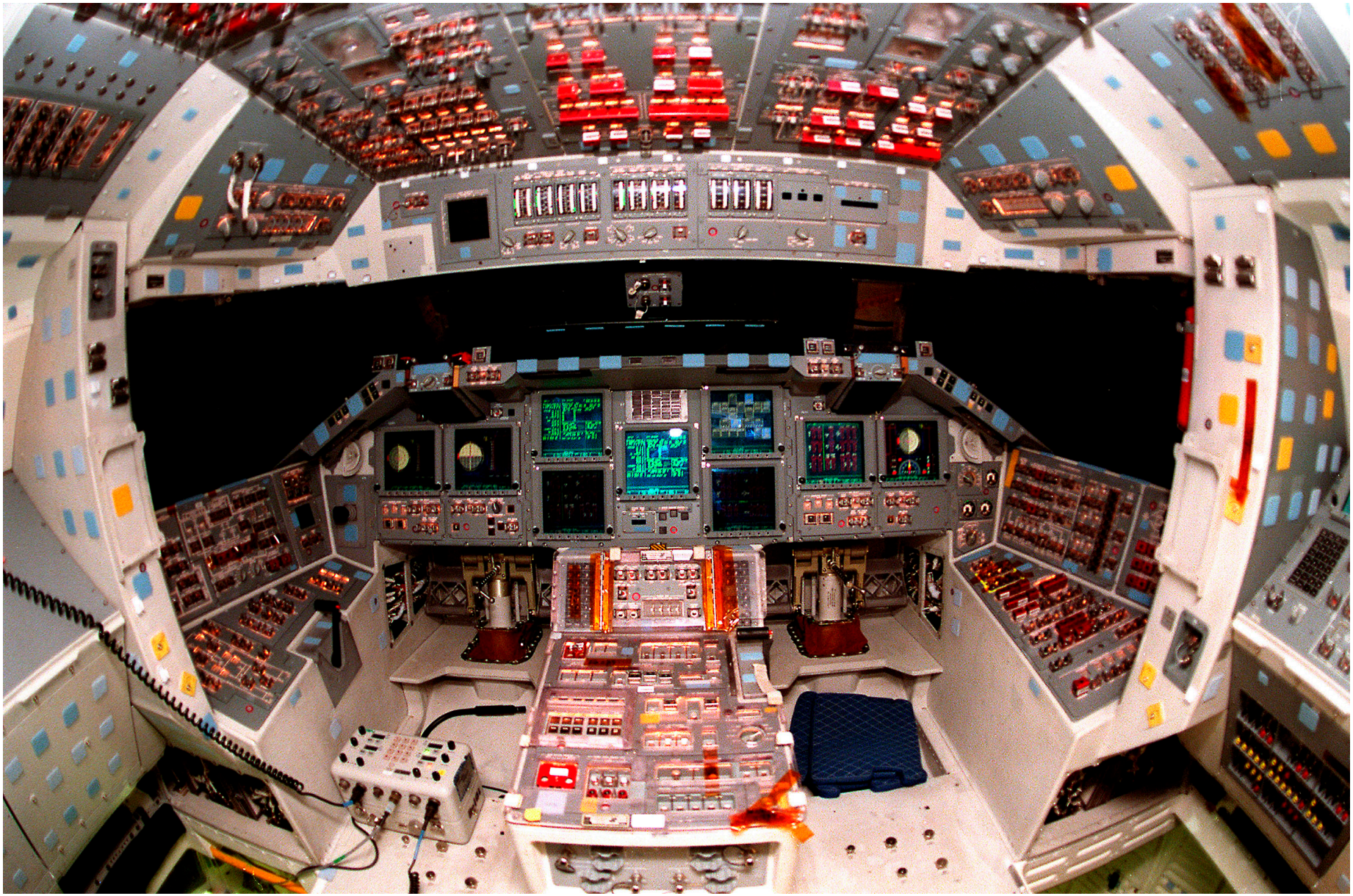
Subset Used in CIS 5000

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

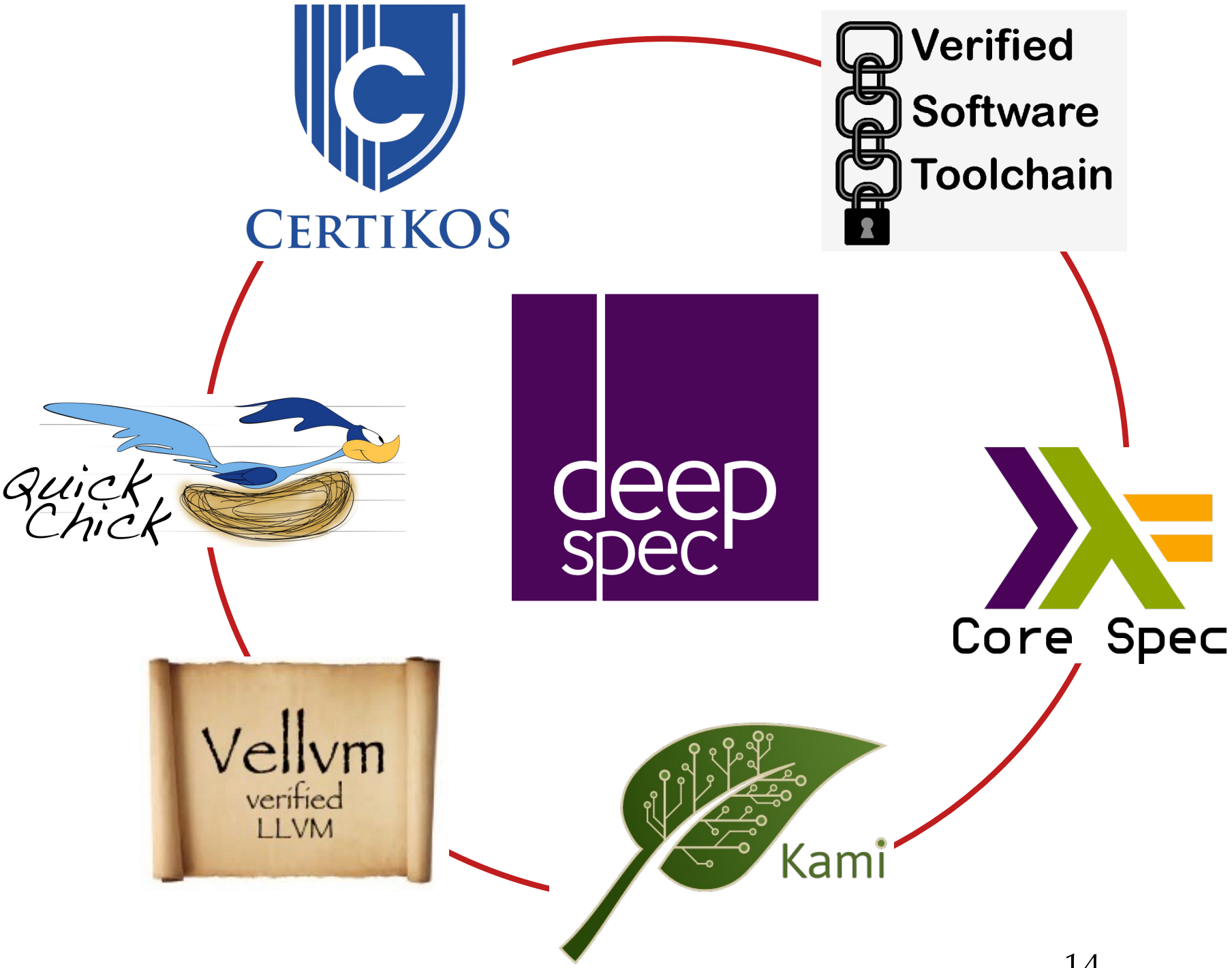
Expeditions Grant
deepspec.org

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

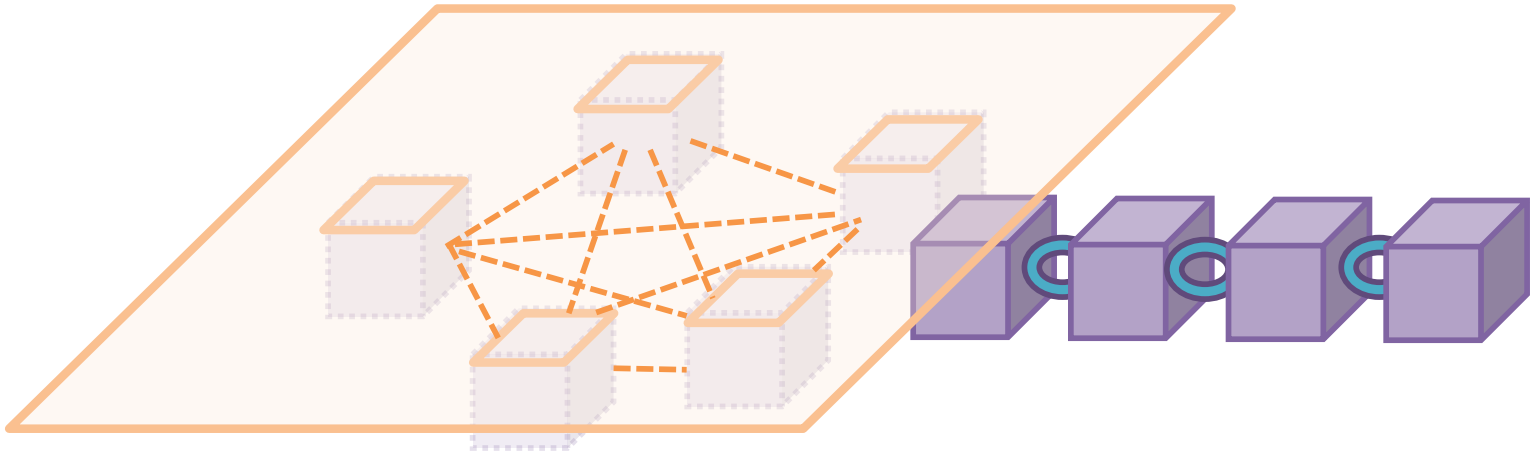
deep spec

Uses deep spec results

CARDANO

Tezos

Smart Contracts

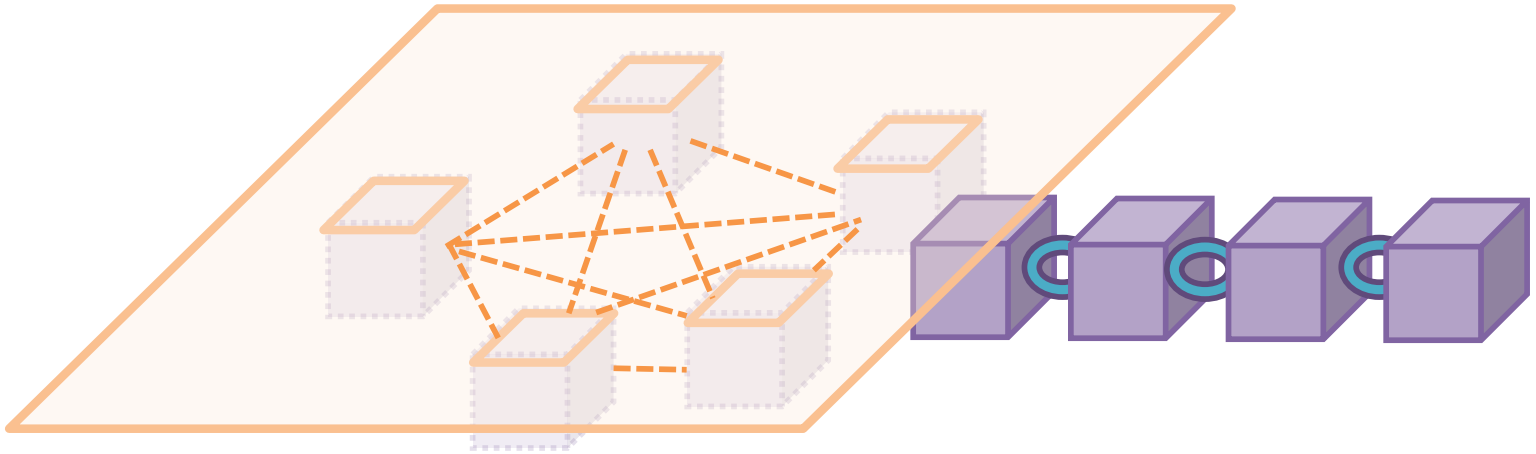


blockchain

Smart Contracts

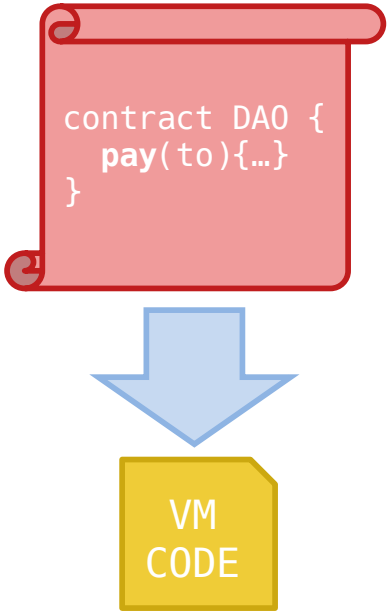
smart contract

- high-level
- human written
- programming language



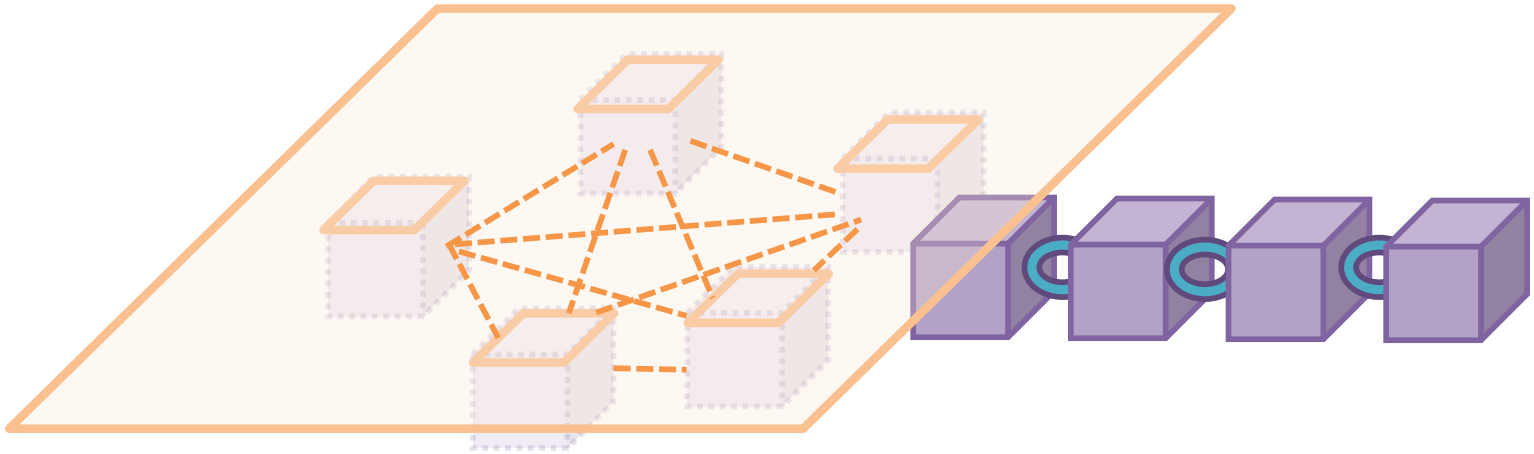
blockchain

Smart Contracts



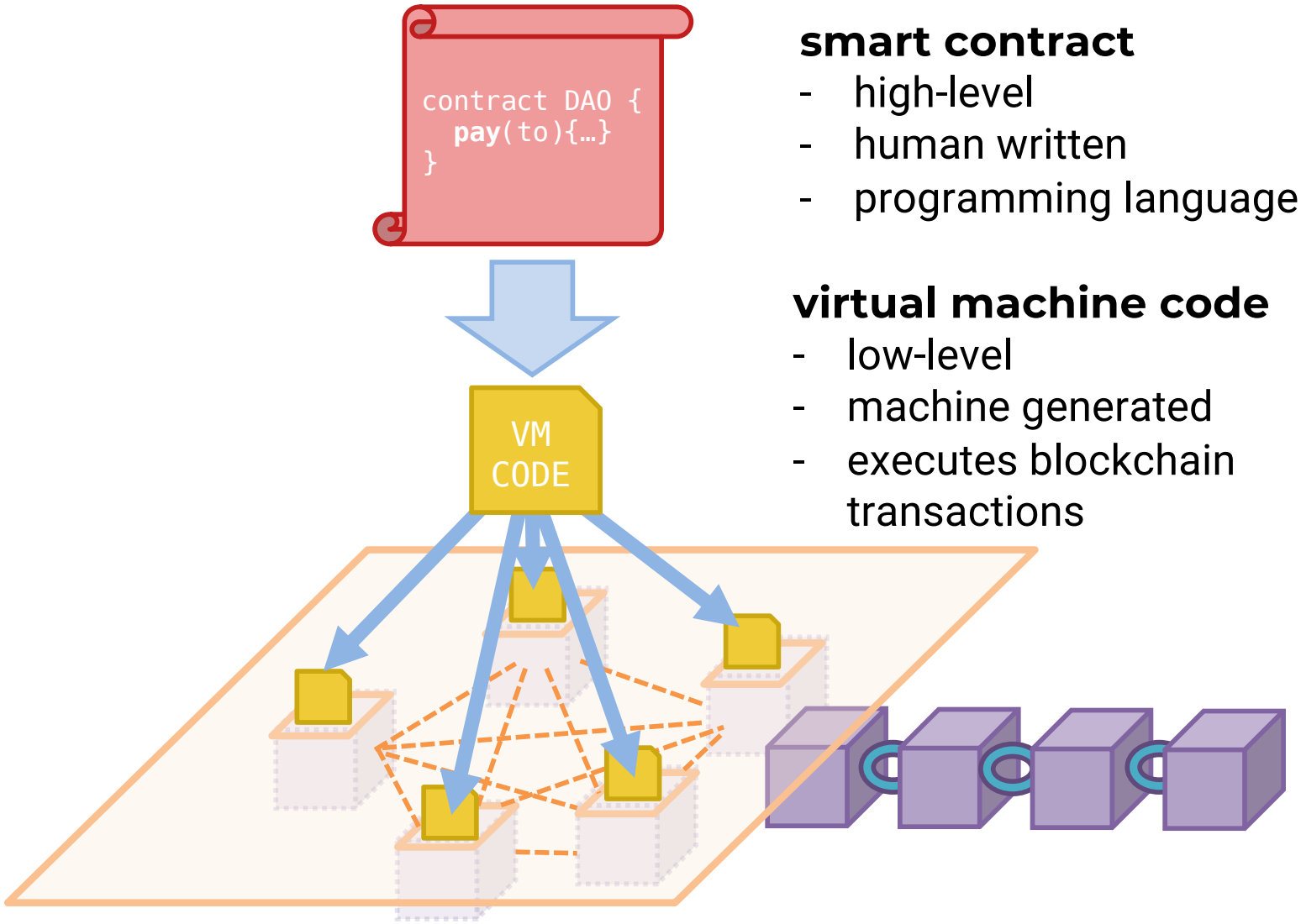
smart contract
- high-level
- human written
- programming language

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



blockchain

Smart Contracts



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

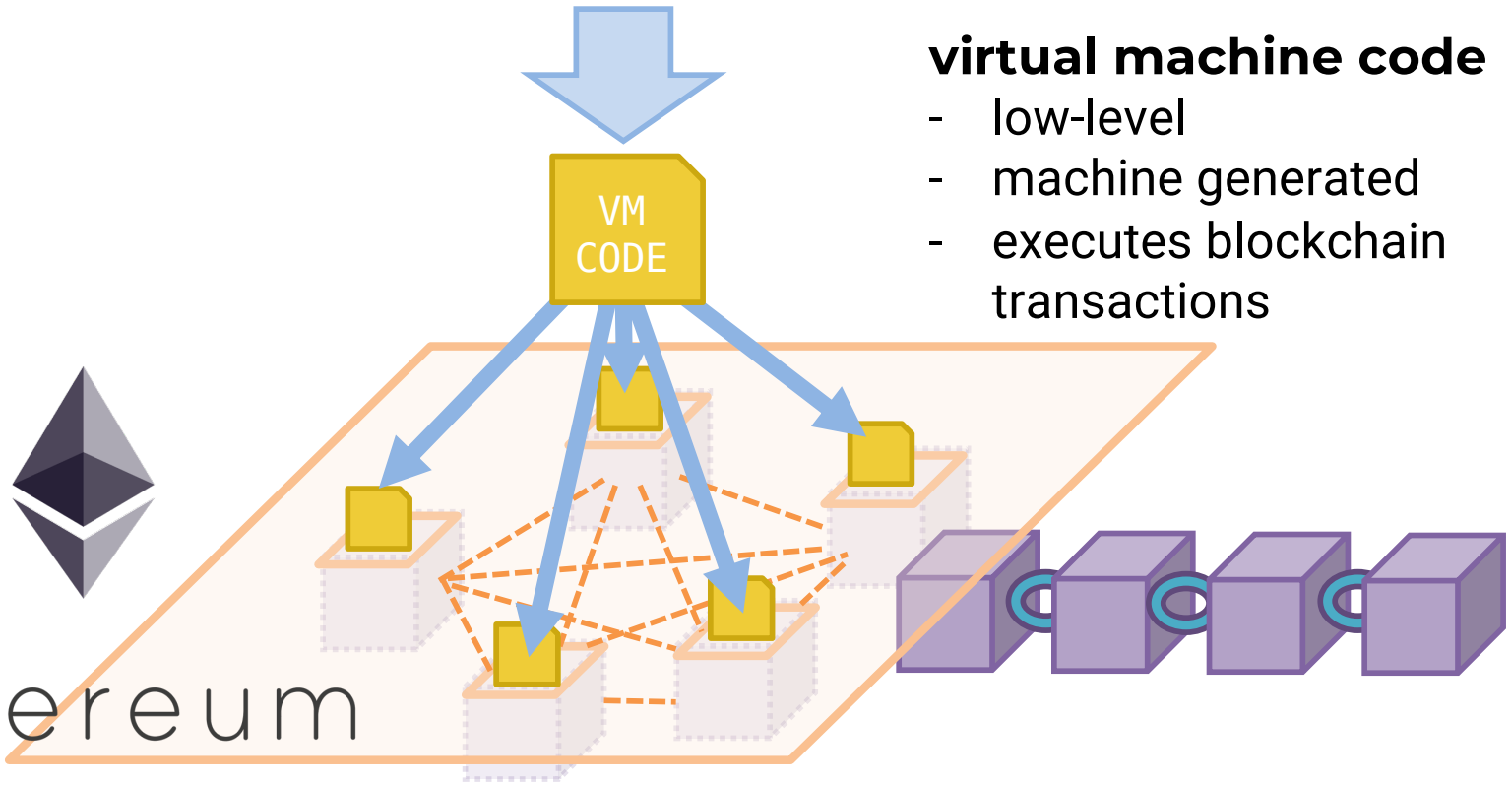
virtual machine code
- low-level
- machine generated
- executes blockchain transactions

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

Smart Contracts

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

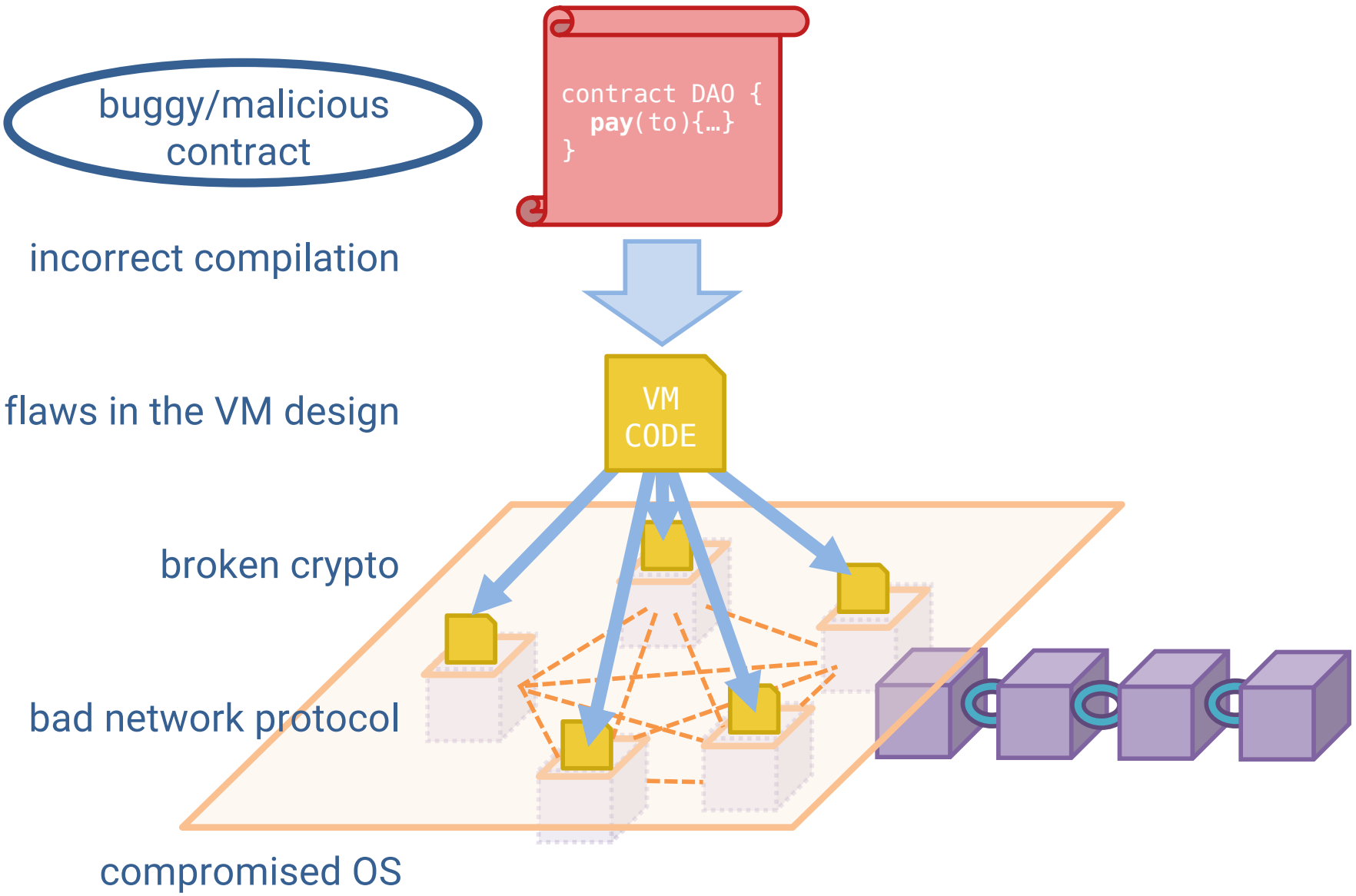
virtual machine code
- low-level
- machine generated
- executes blockchain transactions



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

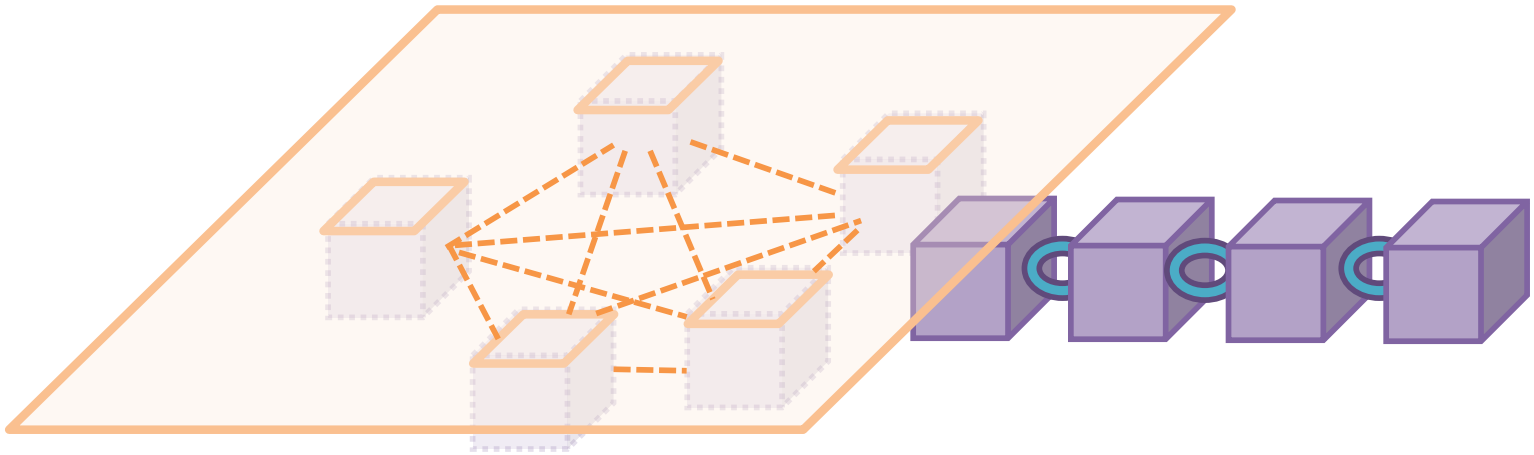
[Atzei, et al. 2017]

```
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

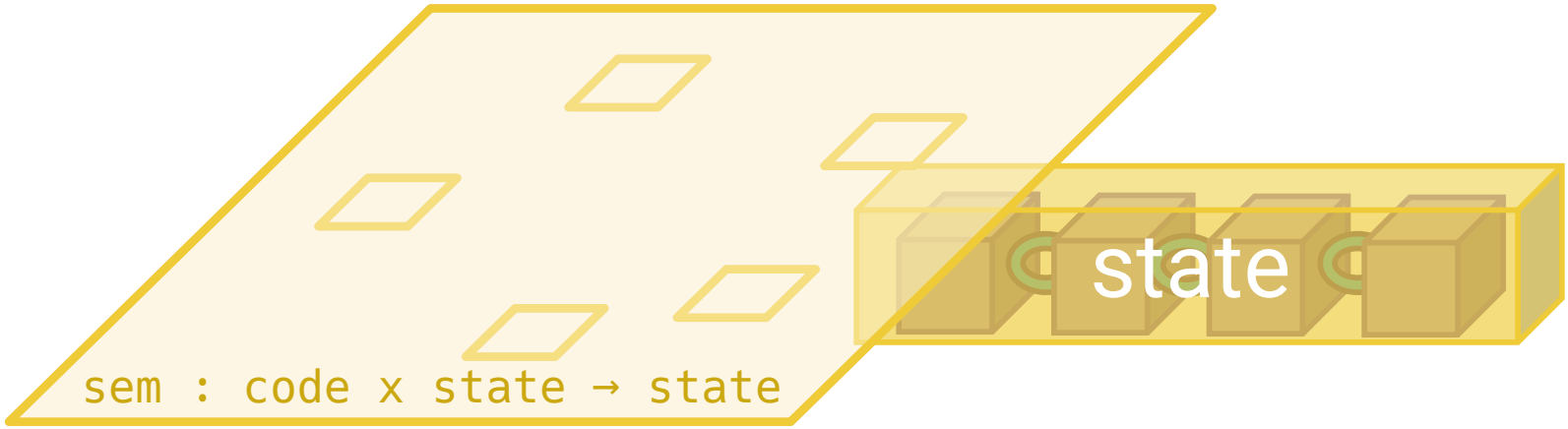


Deep Specification

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

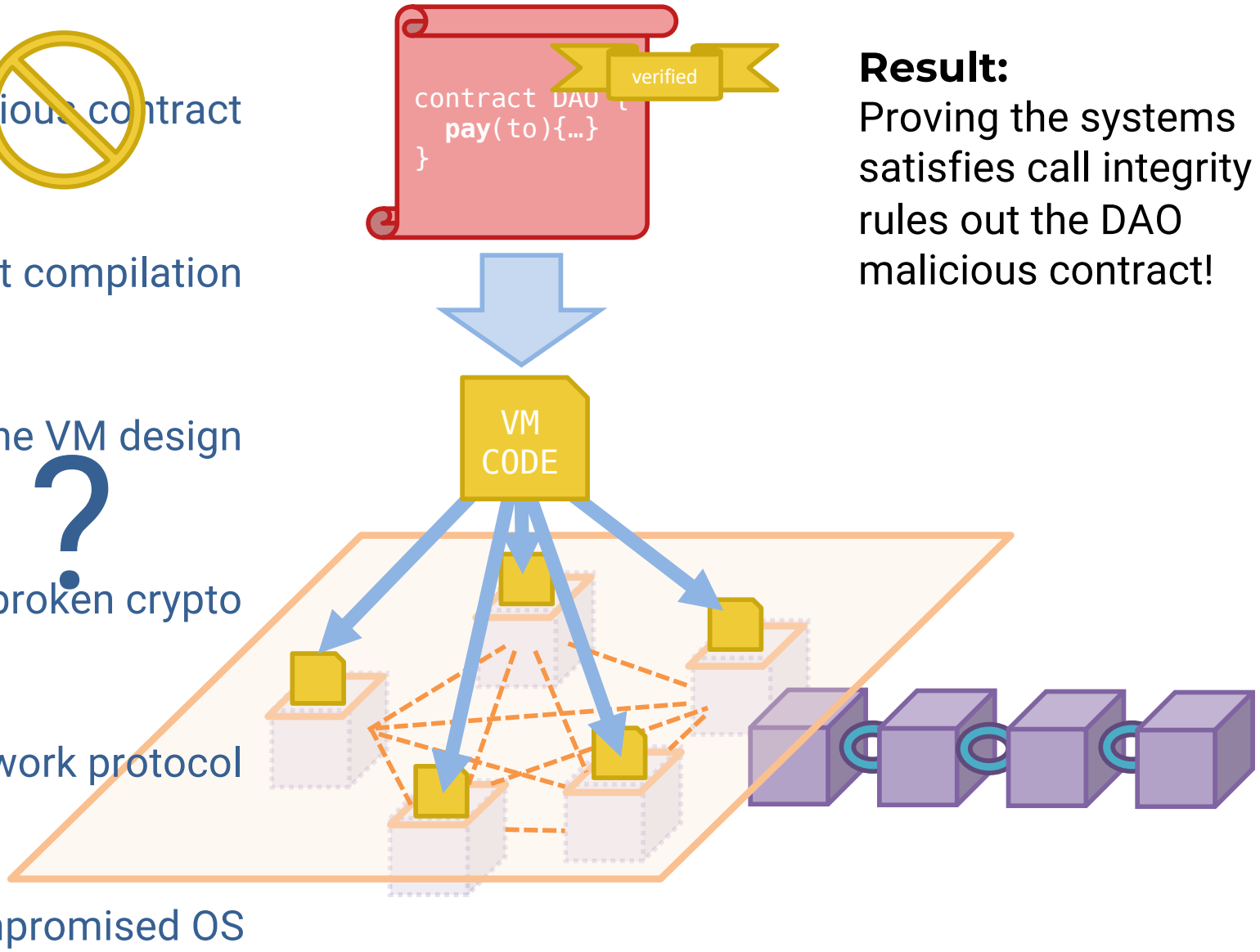
flaws in the VM design

?

broken crypto

bad network protocol

compromised OS



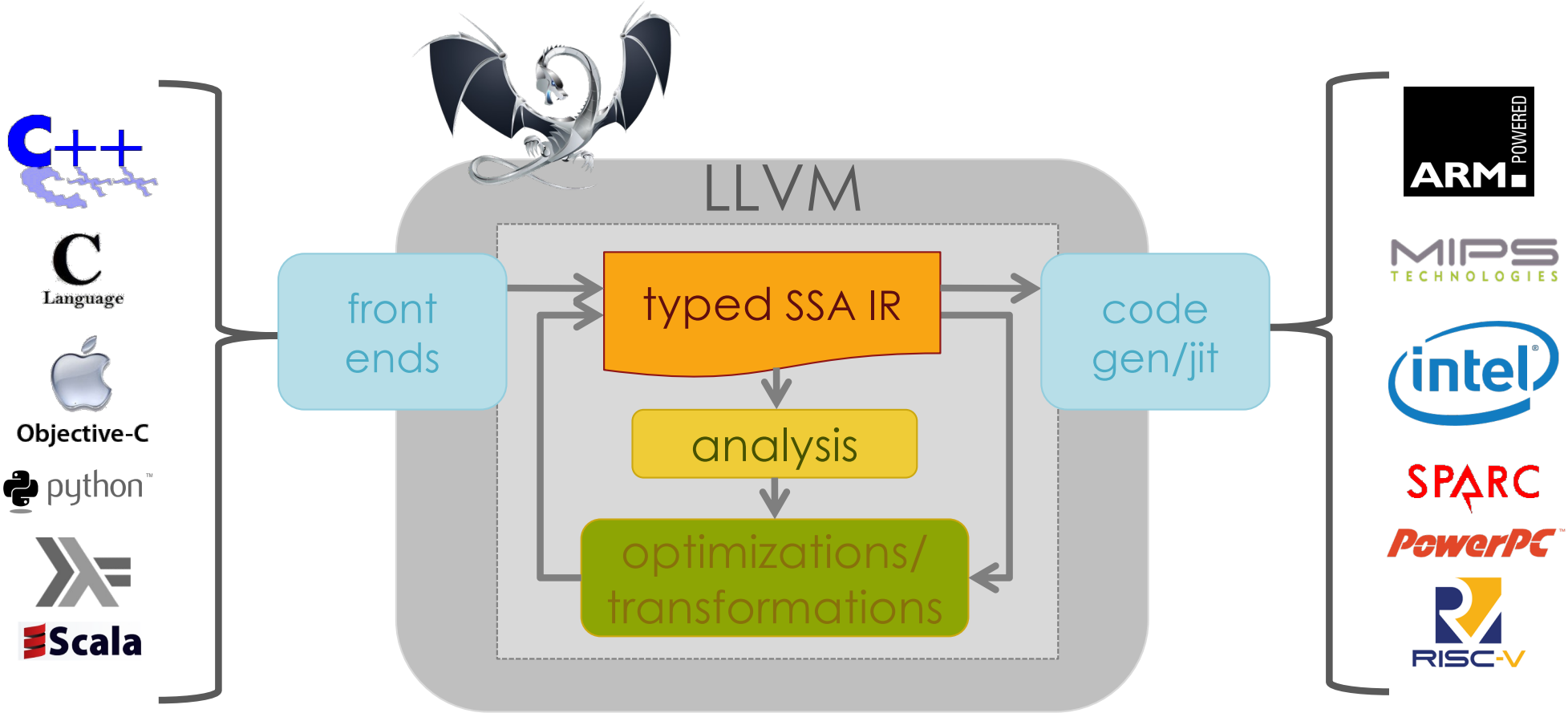
Result:
Proving the systems satisfies call integrity rules out the DAO malicious contract!

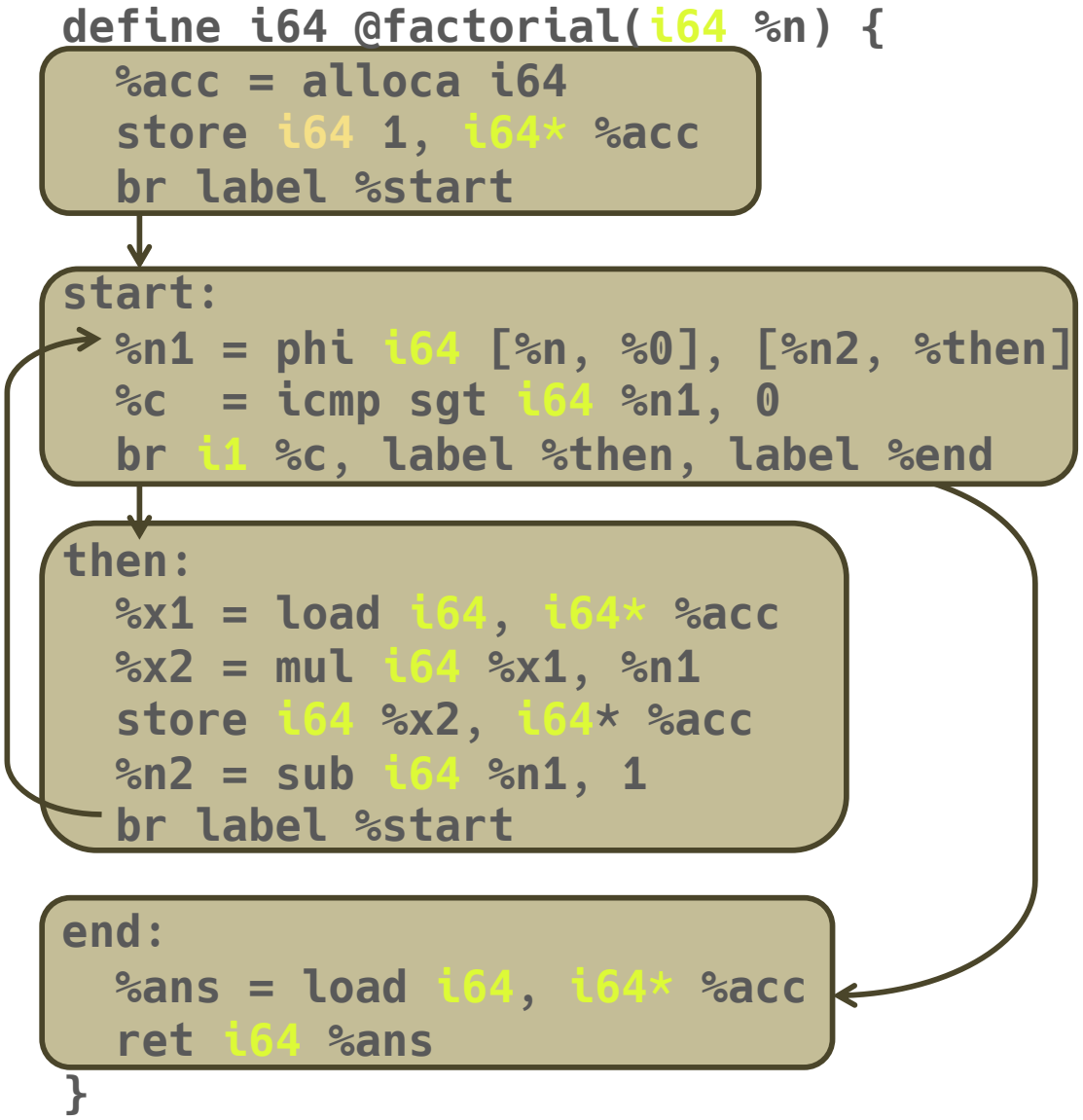
case study

VELLVM – VERIFIED LLVM IR

LLVM Compiler Infrastructure

[Lattner et al.]





LLVM IR

=

Control-flow Graphs:

+ Labeled blocks
+ Straight-line Code
+ Block Terminators
+ Static Single Assignment
+ Phi Nodes

Types:

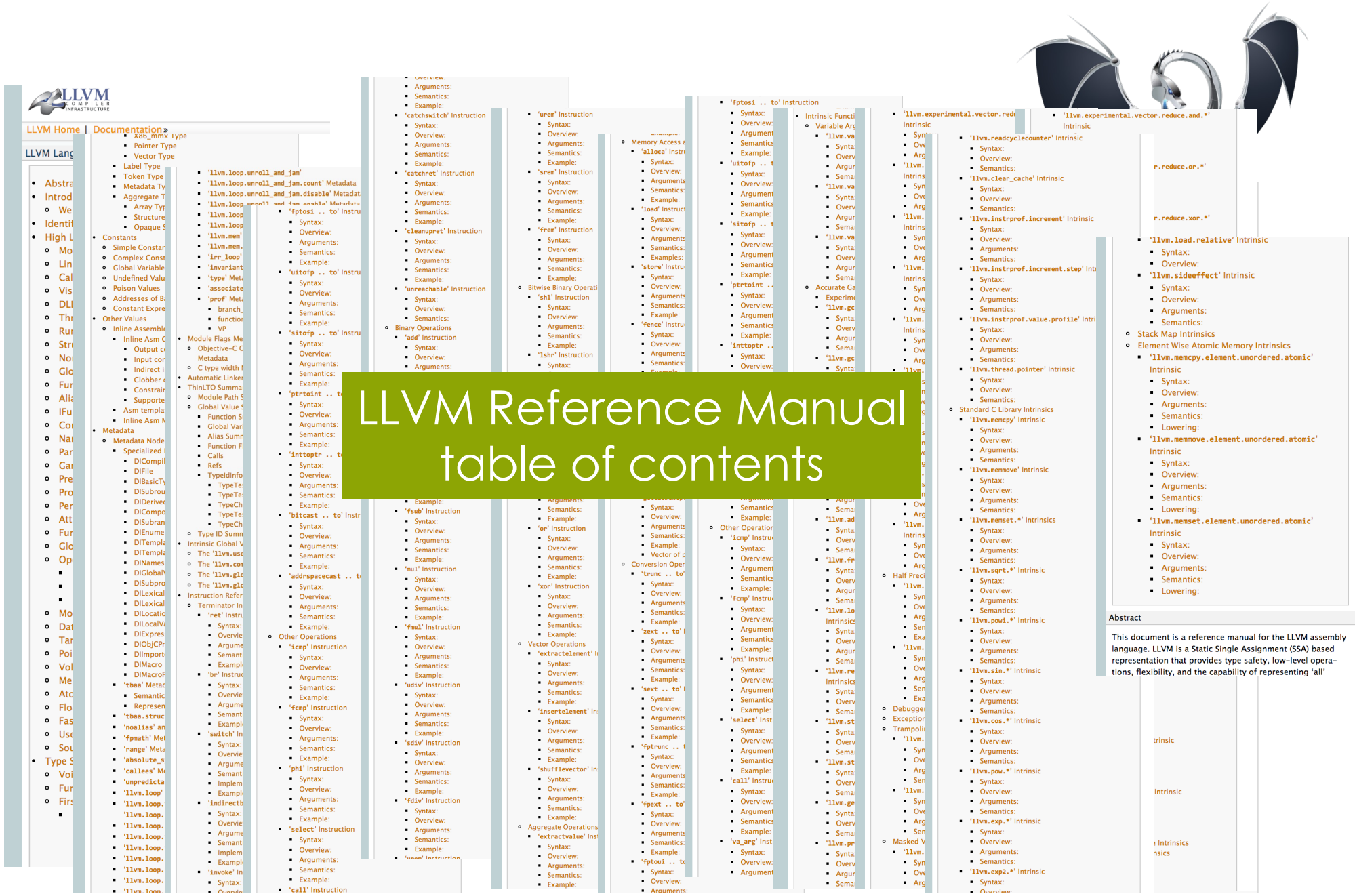
- i64 ⇒ 64-bit integers
- i64* ⇒ pointer

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:

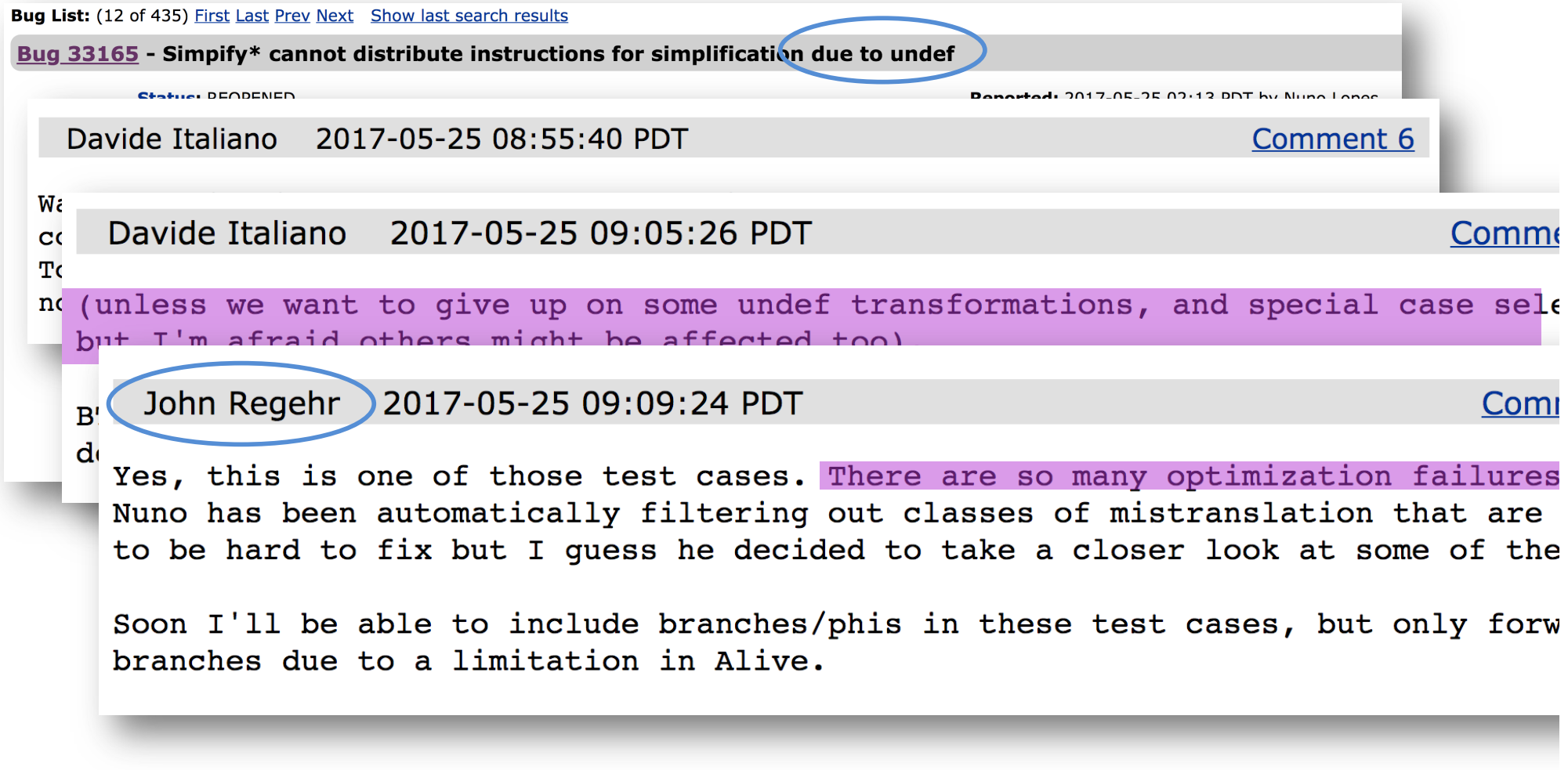
```
%y = mul i8 %x, 2
```

versus:

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

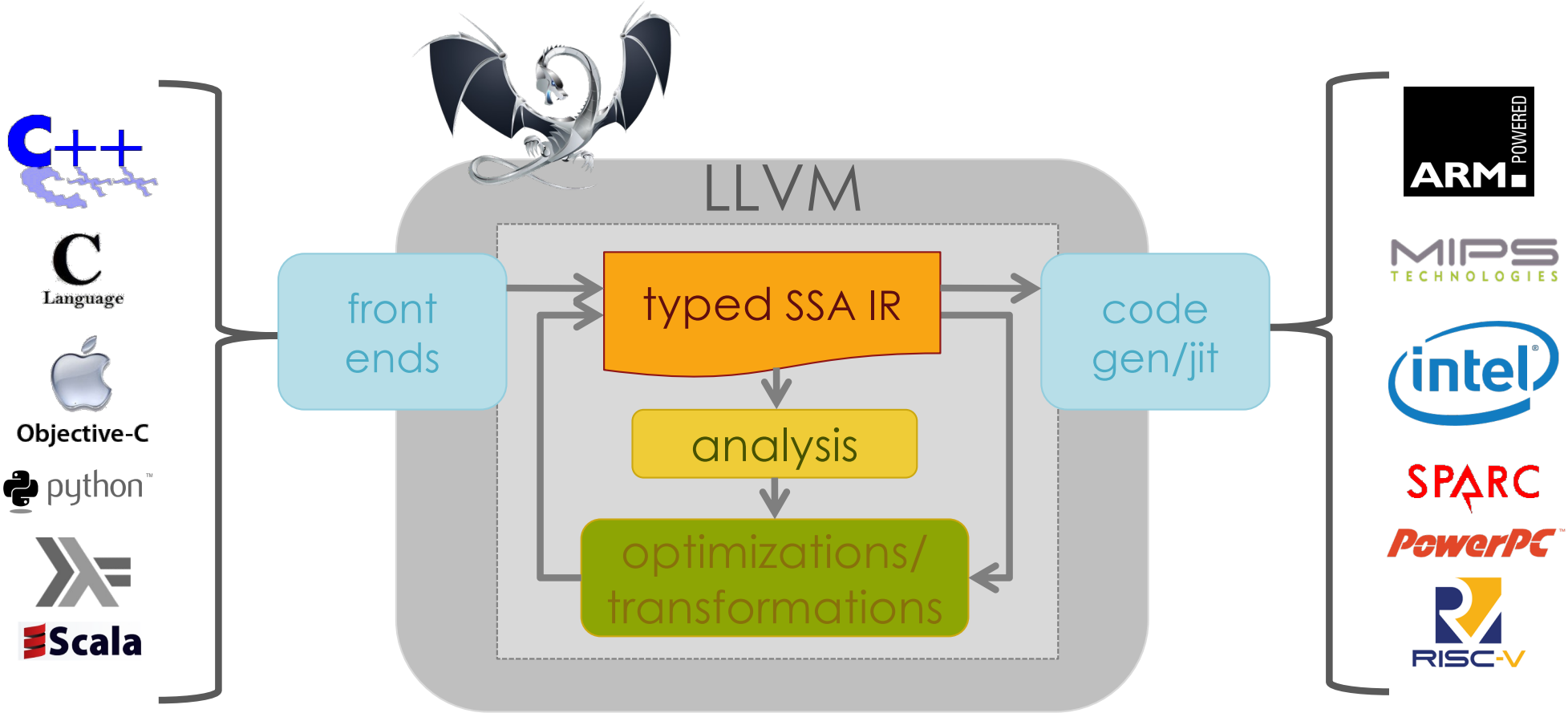
Upshot: if **%x** is **undef**, we can't optimize **mul** to **add**

What's the problem?



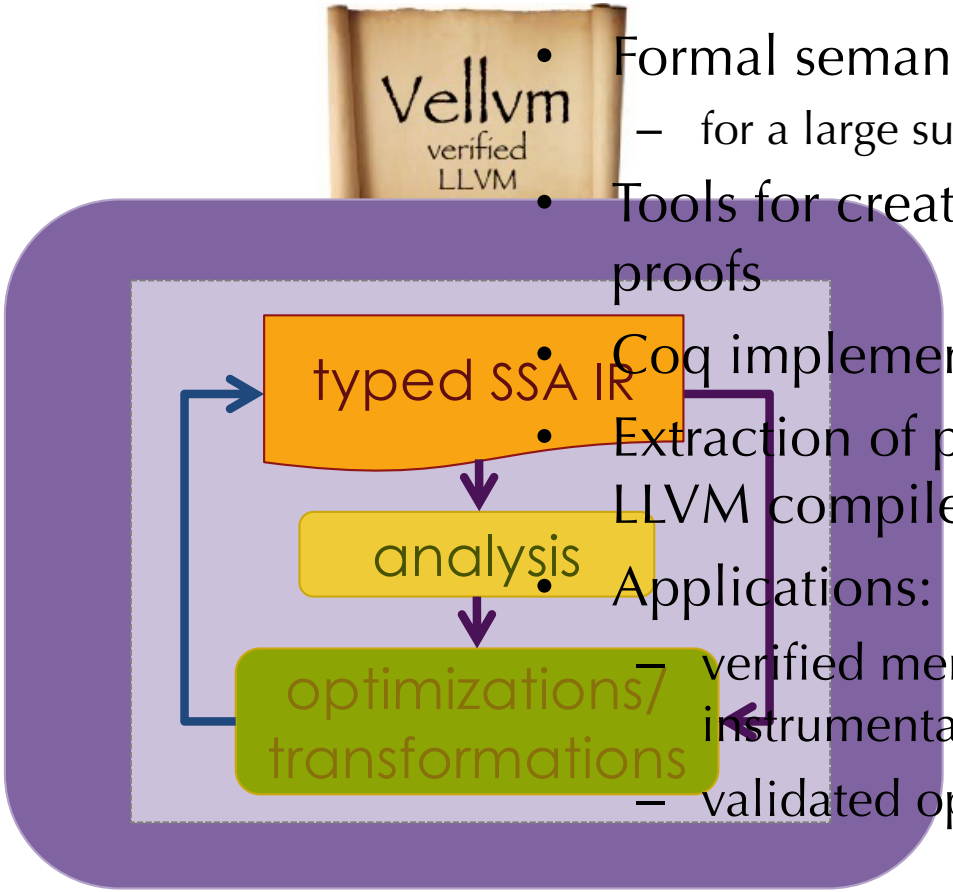
LLVM Compiler Infrastructure

[Lattner et al.]



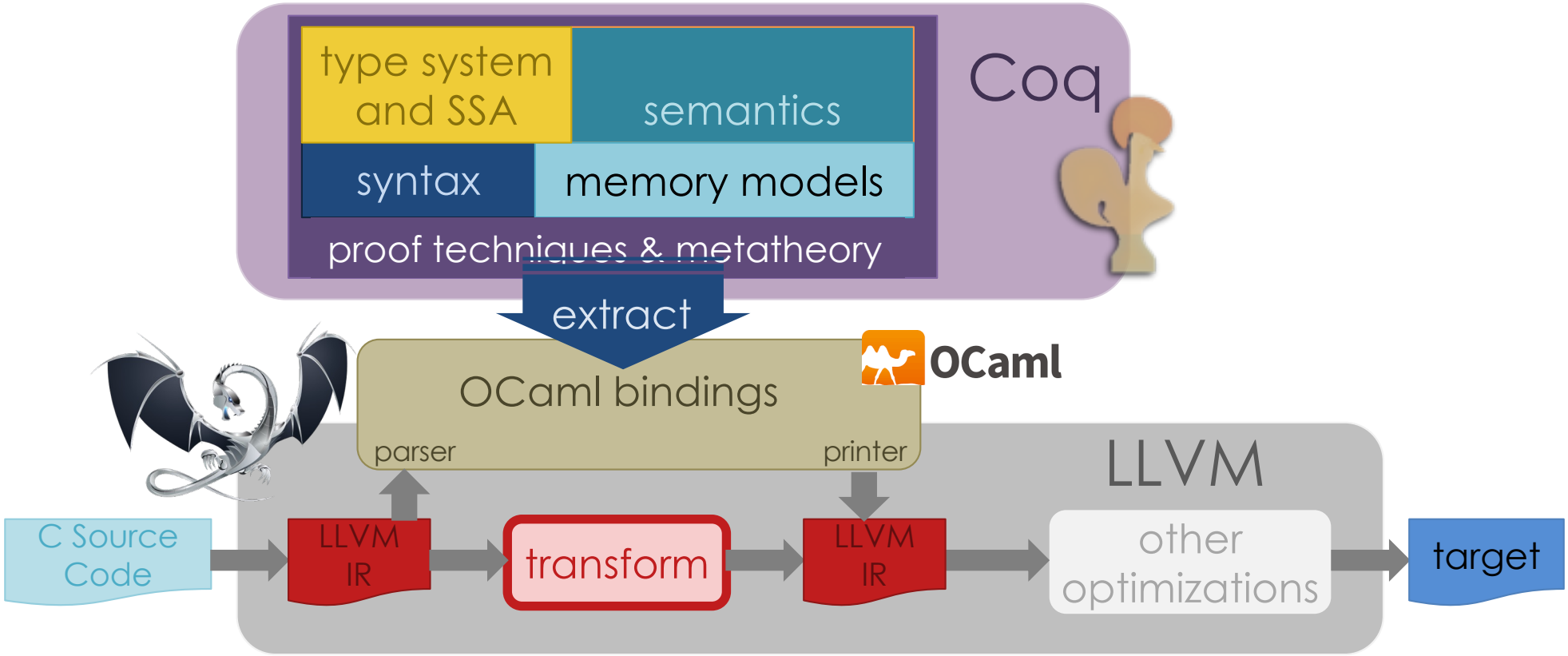
The Vellvm Project

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013, Mansky et al. CAV2015]

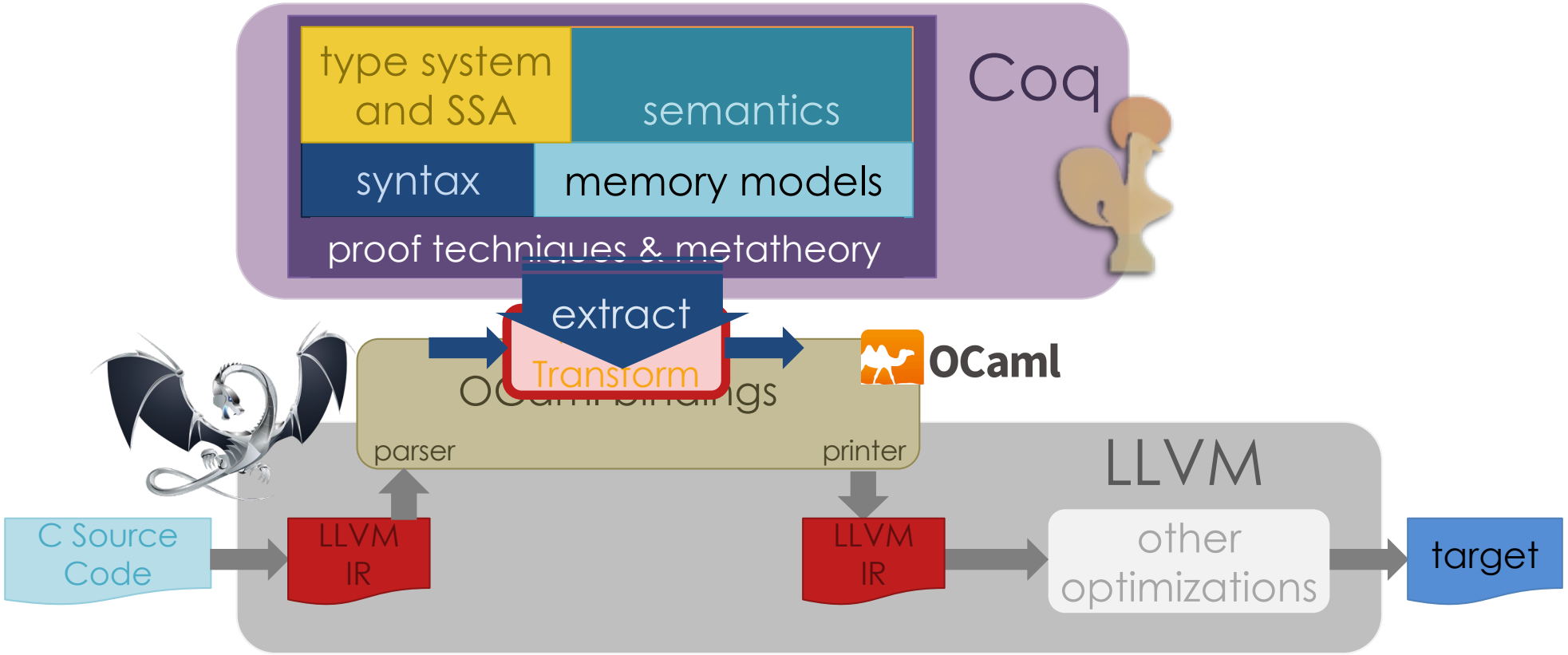


- Formal semantics
  - for a large subset of LLVM
- Tools for creating correctness proofs
- Coq implementation
- Extraction of passes for use with LLVM compiler
- Applications:
  - verified memory safety instrumentation
  - validated optimizations

Vellvm Framework

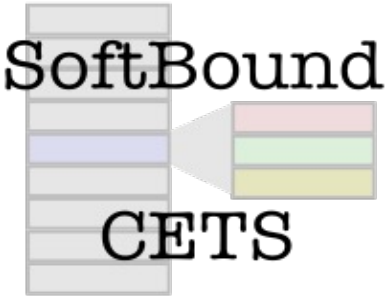


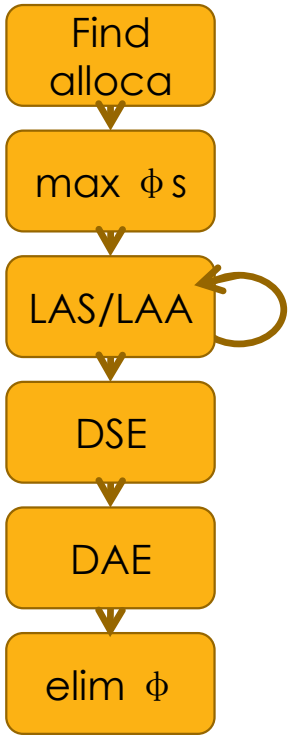
Vellvm Framework



VELLVM – Previous Results

- SoftBound [POPL 2012]
  - Memory Safety
- mem2reg [PLDI 2013]
  - Register promotion, defined in terms of a stack of "micro-optimizations"
- Verified dominator analysis [CPP 2012]
  - Cooper-Harvey-Kennedy Algorithm [2000]
- Better memory models
  - ptrtoint casts [PLDI 2015]
  - modular formalization [CAV 2015]
- Better representation for effectful programs in Coq [POPL2020]
- Improved LLVM IR semantics [ICFP 2021]

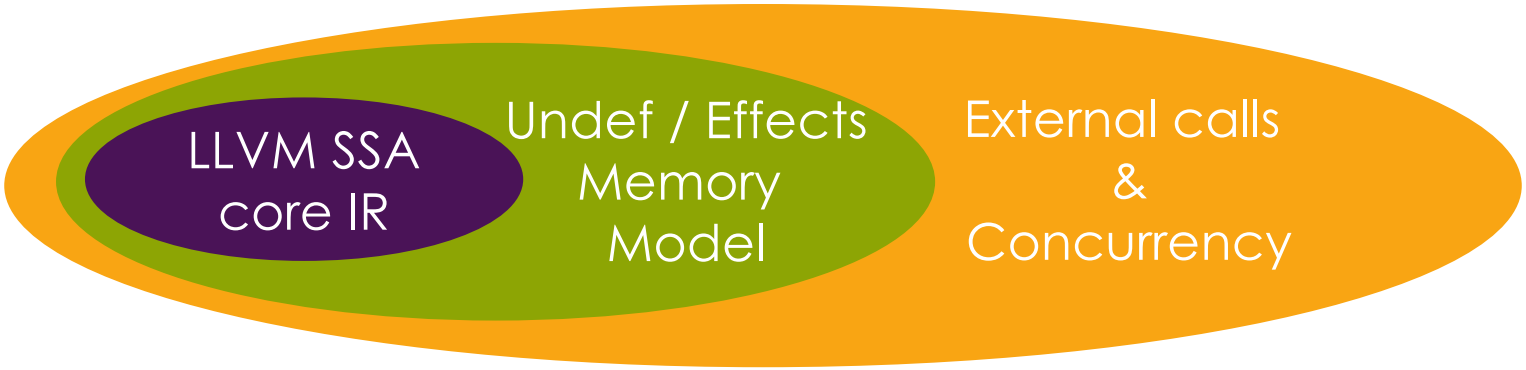




optimizations

Modular LLVM IR Semantics

- Core: LLVM control-flow-graph interpreter
- Memory Model [CAV 15]
  - interprets loads/stores
  - support for casts [PLDI 15]
  - other effects



LLMV Memory Model (simplified)

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

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

output values of the Call event

type of the result provided by the environment

LLVM Interpreter in Coq

```
Definition denote_instr
           (i: (instr_id * instr dtyp)): itree instr_E unit
  match i with
  (* Pure operations *)
  | (IId id, INSTR_Op op) ⇒
    dv ← translate exp_E_to_instr_E (eval_op op) ;;
    trigger (LocalWrite id dv)

  (* Allocation *)
  | (IId id, INSTR_Alloca dt _ _) ⇒
    dv ← trigger (Alloca dt);;
    trigger (LocalWrite id (dvalue_to_uvalue dv))

  (* Load *)
  | (IId id, INSTR_Load _ dt (du,ptr) _) ⇒
    ua ← translate exp_E_to_instr_E (denote_exp (Some du) ptr) ;;
    da ← trigger (pick ua T) ;;
    match da with
    | DVALUE_Poison → raiseUB "Load from poisoned address."
    | _ ⇒ dv ← trigger (Load dt da);;
          (* debug ("Loading: " ++ to_string dv);; *)
          trigger (LocalWrite id dv)
    end
```

interpreter returns an interaction tree with "LLVM" effects.

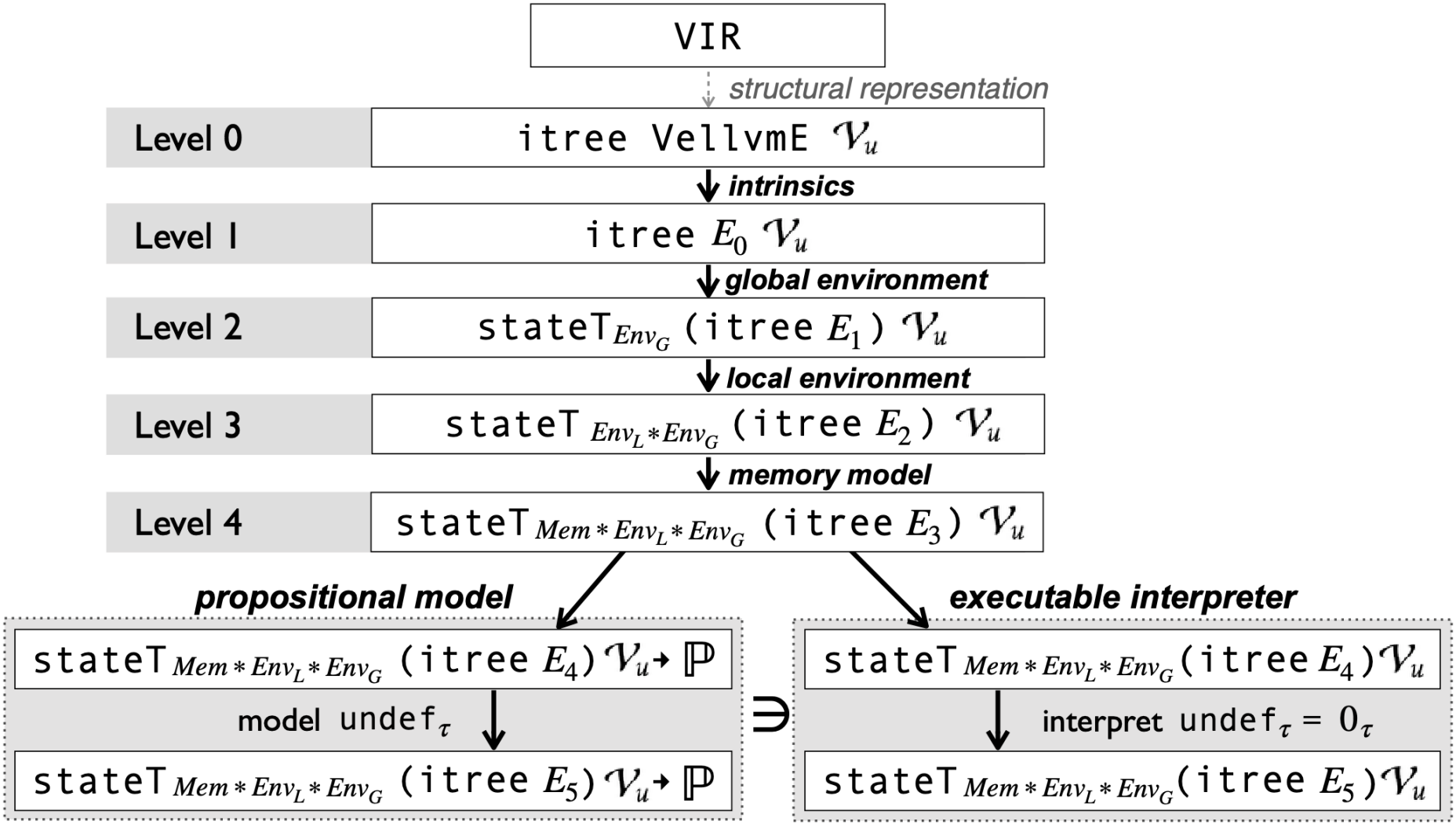
Extract to executable interpreter (Ocaml).

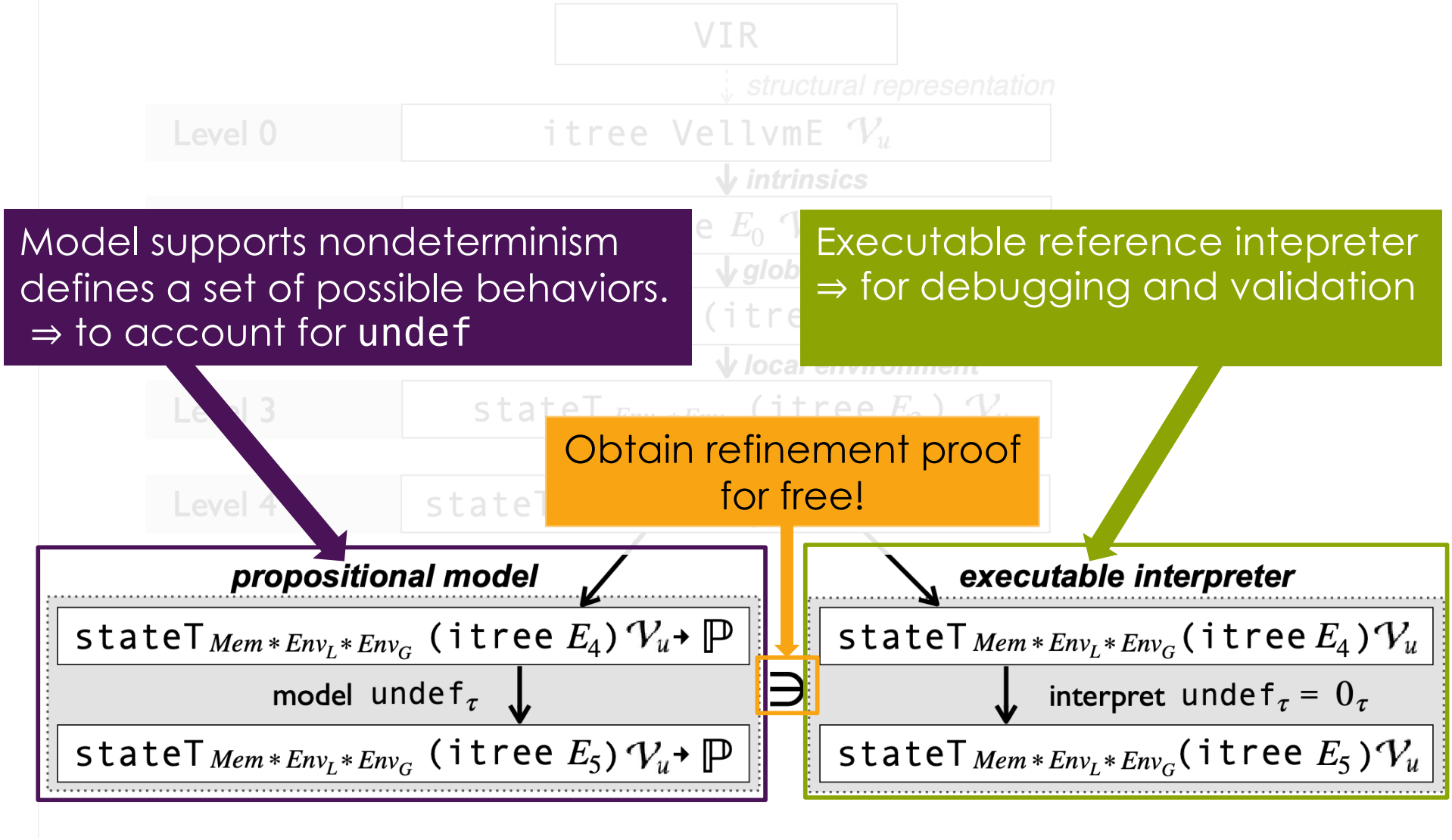
The interpreter "calls out" to the memory model by generating visible effects…

Vellvm Effects

1. External Calls
2. Intrinsics
3. Global Environment
4. Local Environment
5. Stack
6. Memory
7. Nondeterminism
8. Undefined Behavior
9. (Debugging)

Each layer of effects can be interpreted separately, making proofs *modular* and changes orthogonal.


VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
Level I
itree E_0 $\mathcal{V}_u$
global environment
Level 2
stateT$_{Env_G}$ (itree E_1) $\mathcal{V}_u$
local environment
Level 3
stateT $_{Env_L * Env_G}$ (itree E_2) $\mathcal{V}_u$
memory model
Level 4
stateT $_{Mem * Env_L * Env_G}$ (itree E_3) $\mathcal{V}_u$
propositional model
executable interpreter
stateT $_{Mem * Env_L * Env_G}$ (itree E_4) $\mathcal{V}_u \to \mathbb{P}$
model undef$_\tau$
stateT $_{Mem * Env_L * Env_G}$ (itree E_5) $\mathcal{V}_u \to \mathbb{P}$
∋
stateT $_{Mem * Env_L * Env_G}$ (itree E_4) $\mathcal{V}_u$
interpret undef$_\tau$ = 0_τ
stateT $_{Mem * Env_L * Env_G}$ (itree E_5) $\mathcal{V}_u$


Model supports nondeterminism
defines a set of possible behaviors.
⇒ to account for undef
Executable reference intepreter
⇒ for debugging and validation
Obtain refinement proof
for free!
propositional model
executable interpreter
stateT Mem * Env_L * Env_G (itree E_4) V_u → ℙ
model undef_τ
stateT Mem * Env_L * Env_G (itree E_5) V_u → ℙ
∋
stateT Mem * Env_L * Env_G (itree E_4) V_u
interpret undef_τ = 0_τ
stateT Mem * Env_L * Env_G (itree E_5) V_u

Compositionality

DENOTATIONAL SEMANTICS

Compositional LLVM Semantics

$[\![-]\!]_{expr}$ expressions

$[\![-]\!]_{instr}$ instructions

$[\![-]\!]_{block}$ blocks

$[\![-]\!]_{cfg}$ control-flow graphs

$[\![-]\!]_{llvm}$ programs
(mutually recursive functions)

Meaning built up by induction on the structure of the syntax.
- open programs
- fixpoint combinators
- pure monadic computations

Operational Semantics
- not defined purely by induction on syntax
- extra "bookkeeping": evaluation contexts, program counters, etc.
- ☹

Compositional Reasoning

$$\frac{\texttt{outputs}(cfg_2) \cap \texttt{inputs}(cfg_1) = \emptyset \quad to \notin \texttt{inputs}(cfg_1)}{[\![cfg_1 +\!\!+ cfg_2]\!]_{\text{cfg}}\ (f, to) \approx [\![cfg_2]\!]_{\text{cfg}}\ (f, to)}\ \textsc{SubCFG1}$$

$$\frac{}{\begin{array}{c}[\![cfg_1 +\!\!+ cfg_2]\!]_{\text{cfg}}\ (f, to) \approx [\![cfg_1]\!]_{\text{cfg}}\ (f, to)\ ;; \\ \texttt{match}\ x\ \texttt{with} \mid \texttt{inl}\ f\ to \Rightarrow [\![cfg_1 +\!\!+ cfg_2]\!]_{\text{cfg}}\ f\ to \\ \mid \texttt{inr}\ v \Rightarrow \texttt{ret}\ v\end{array}}\ \textsc{SubCFG2}$$

$$\frac{\texttt{independent\_flows}\ cfg_1\ cfg_2 \quad to \in \texttt{inputs}(cfg_i)}{[\![cfg_1 +\!\!+ cfg_2]\!]_{\text{cfg}}\ (f, to) \approx [\![cfg_i]\!]_{\text{cfg}}\ (f, to)}\ \textsc{Flow}$$

Rules to reason about control-flow graph composition.

Lemmas about "contextual program equivalences".

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

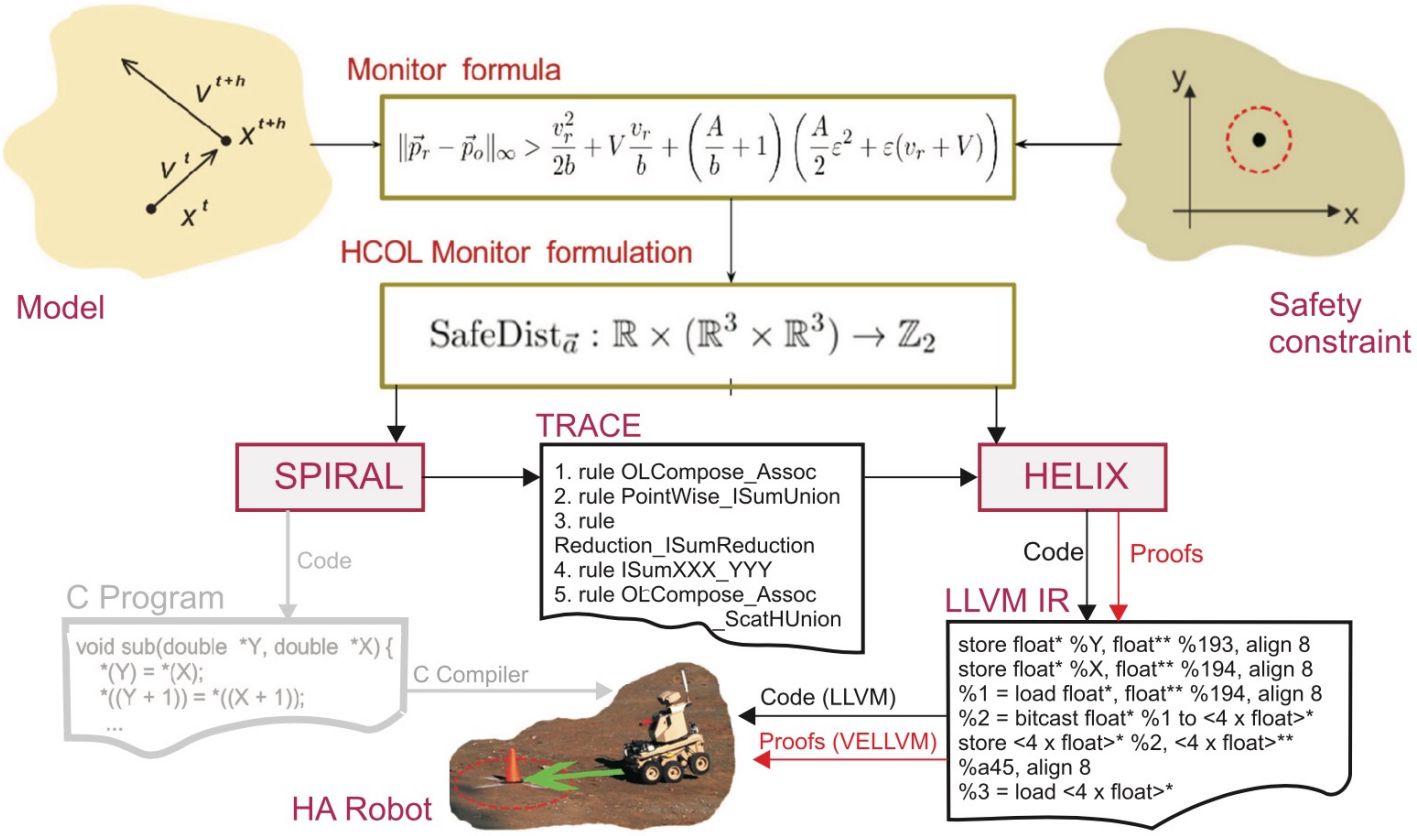
Deep Specifications

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

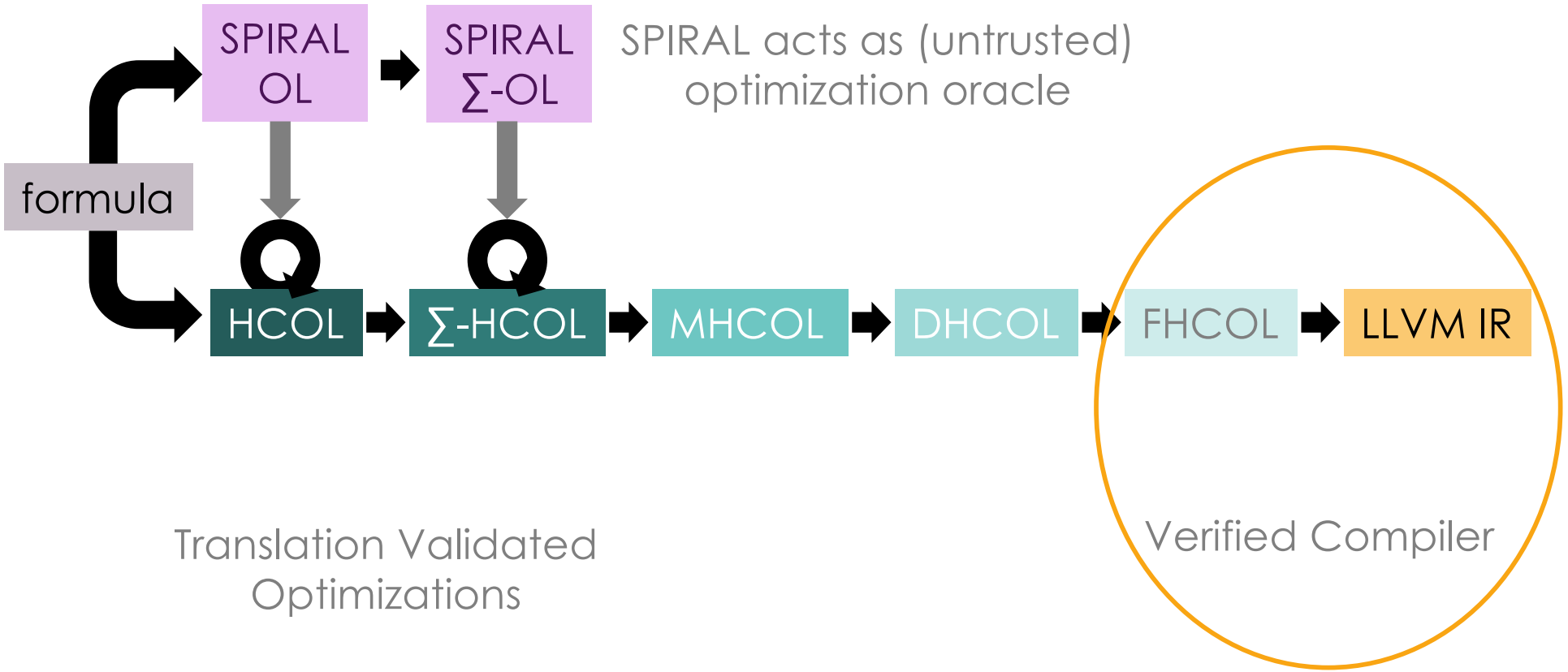
SPIRAL / HELIX

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

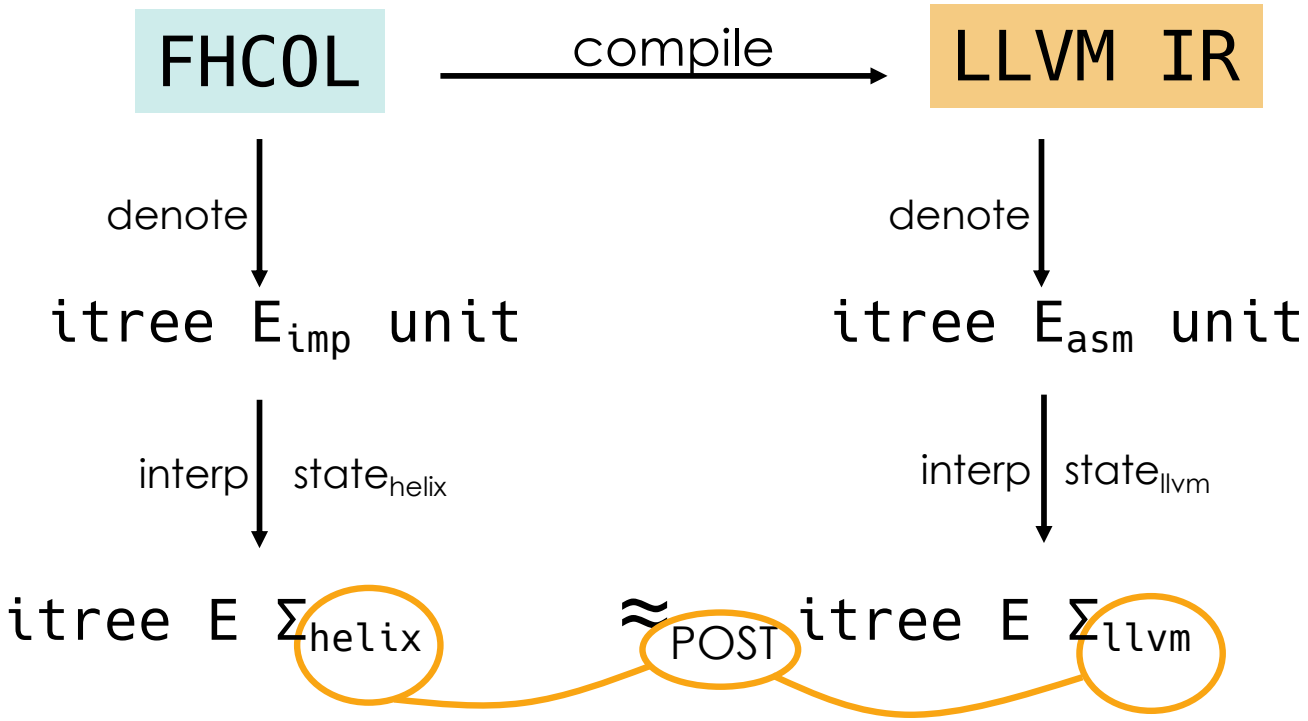
DSL for high-performance numerical computing.



HELIX Compilation Pipeline



Compiler Correctness Proof



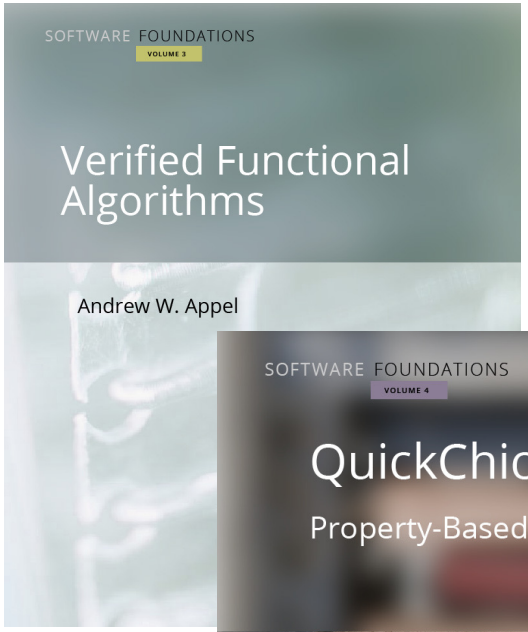
WHAT NEXT?

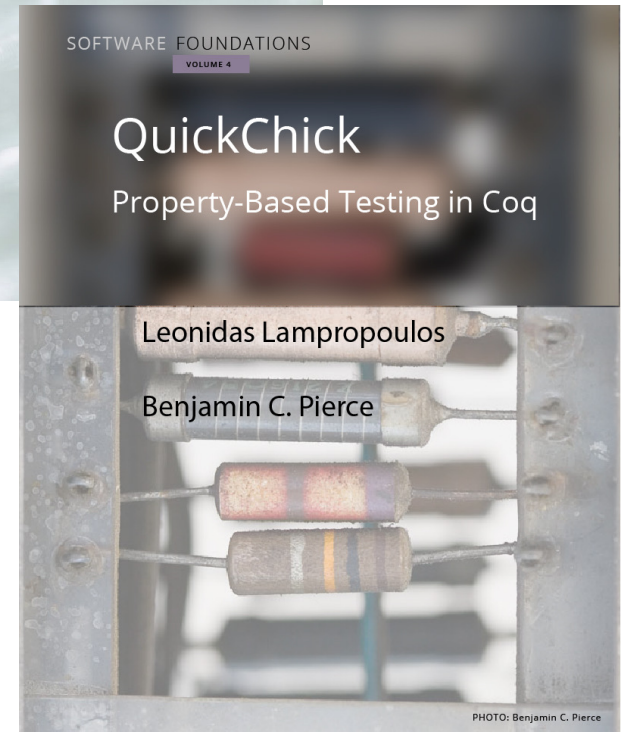
After CIS 5000

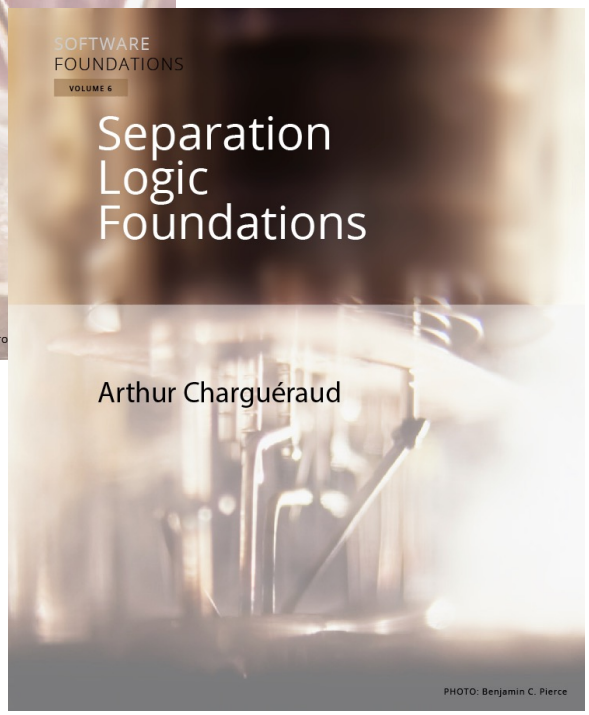
- **Research Conferences:**
  - Certified Programs and Proofs (CPP)
  - Interactive Theorem Proving (ITP)
  - Theorem Proving and Higher-order Logics (TPHOLS)
  - Principles of Programming Languages (POPL)
  - Programming Languages Design and Implementation (PLDI)
  - International Conference on Functional 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





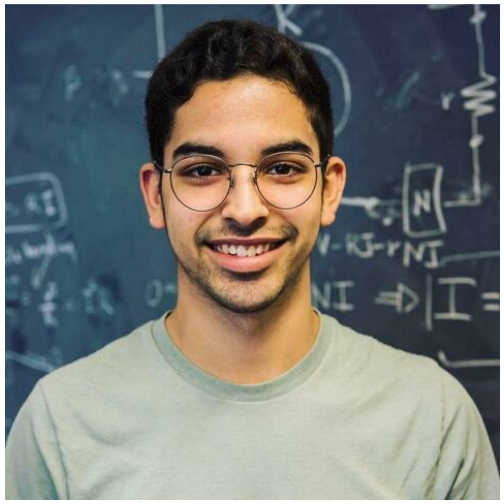


Thanks!

- To our three fantastic TAS:

Jessica Shi

Adam Stein



Joey Velez-Ginorio

And, thanks to all of you!