Modular Semantics ▷ (and Meta-theory) for LLVM IR

Cambium Seminar

Inria

Irene Yoon

12 / 04 / 2024

Joint work with collaborators at...

Penn
University of Pennsylvania

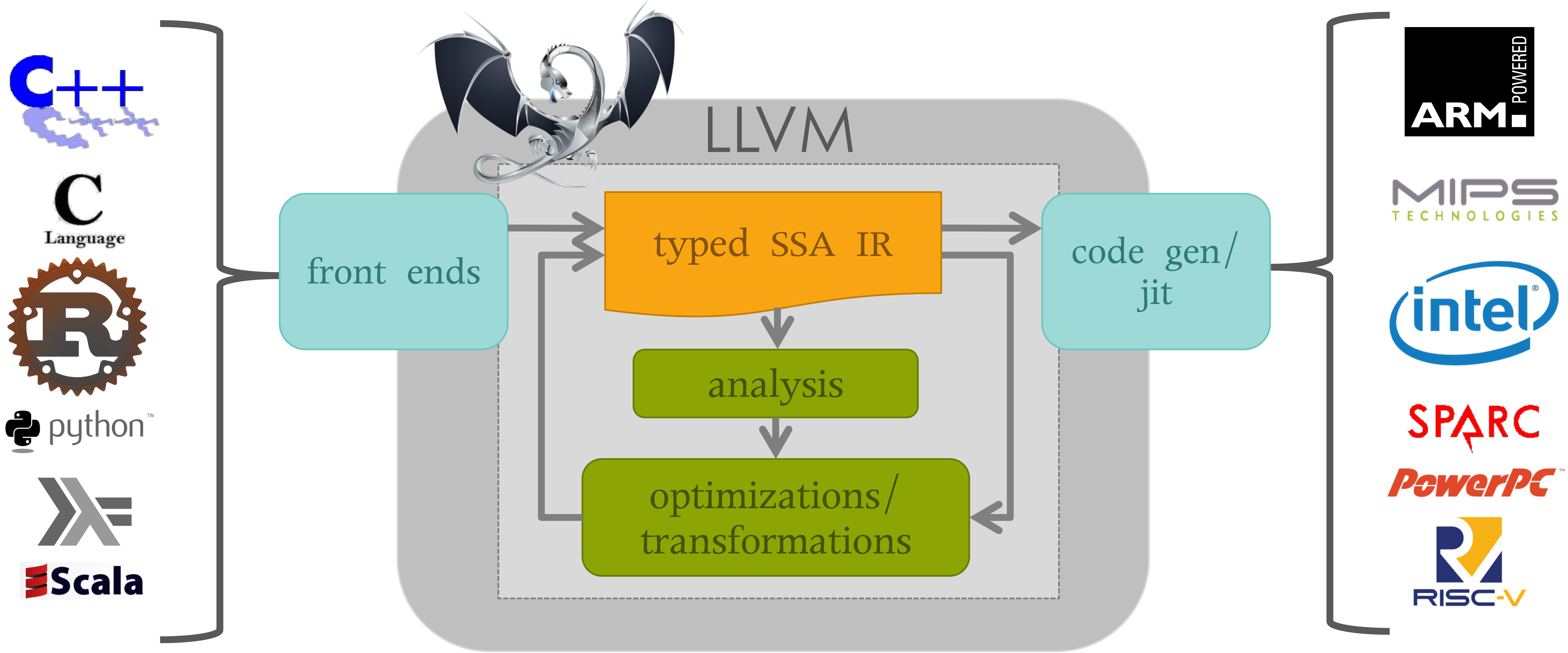
Inria

CMU

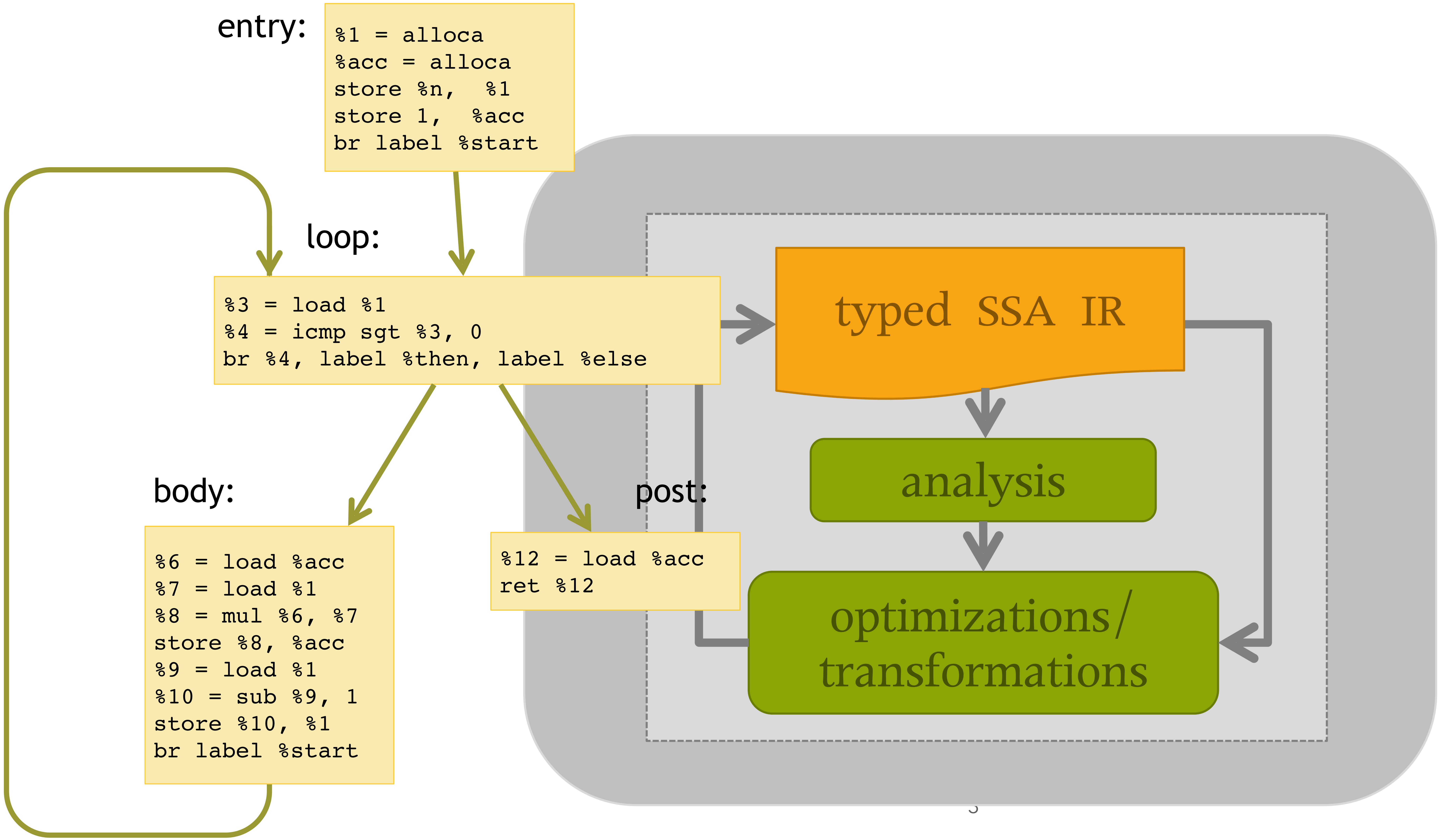
MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

LLVM Compiler Infrastructure [Lattner et al.]

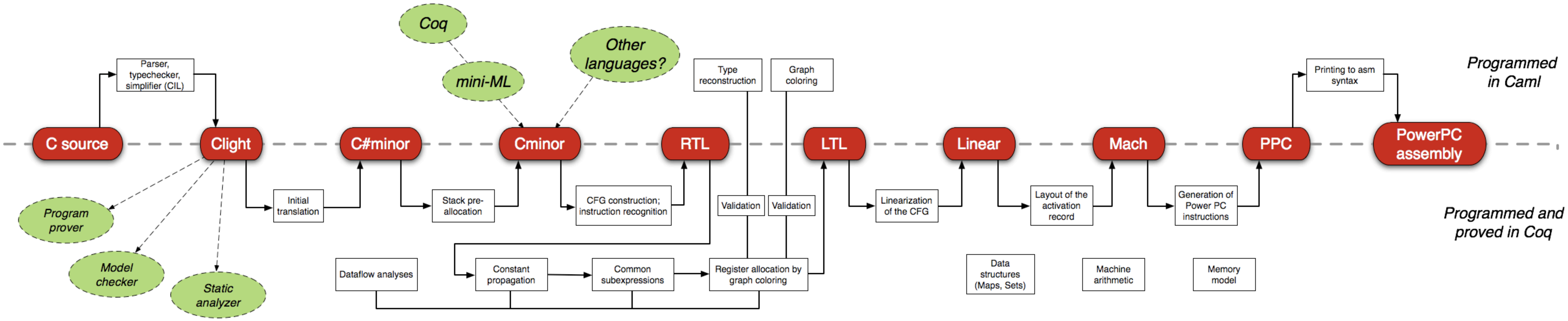
A modular and reusable infrastructure for compiler pipelines



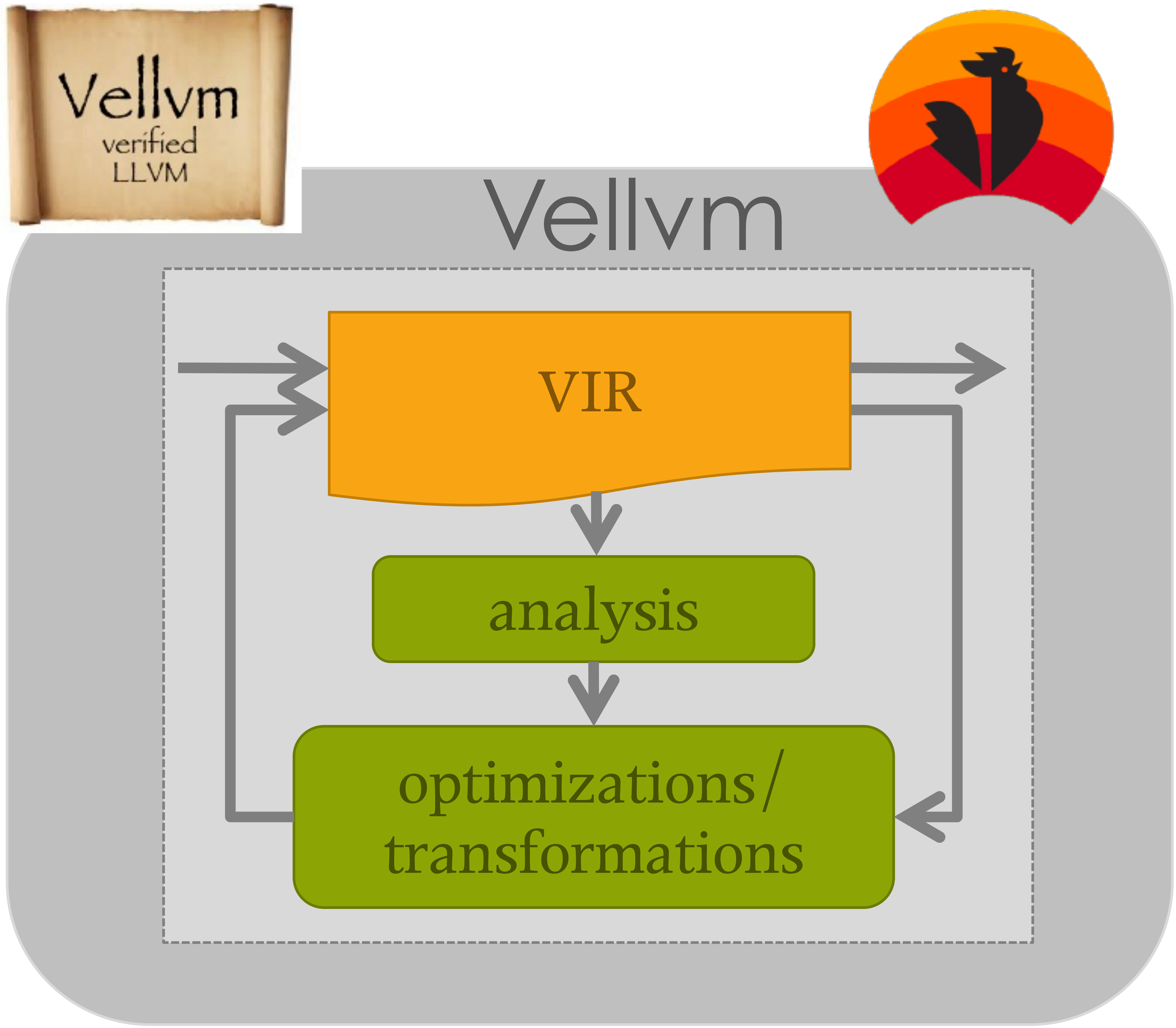
Understanding LLVM Intermediate Representation (IR)



CompCert [Leroy et al.]



The Vellvm Project ("Vellvm 1.0")



[Zhao and Zdancewic - CPP 2012]
Verified computation of dominators

[Zhao et al. - POPL 2012]
Formal semantics of IR + verified SoftBound

[Zhao et al. - POPL 2013]
Verification of (v)mem2reg!

https://github.com/vellvm/vellvm-legacy

A success, but monolithic

$$\boxed{G \vdash pc, mem \rightarrow pc', mem'}$$

Operational Semantics

- A single relation encompasses all aspects of the semantics
- The relation is propositional (and Coq "Prop" cannot be extracted), thus the semantics cannot be extracted
- Classic simulation proofs

Testing matters!

You can model the lambda-calculus as a mathematician.

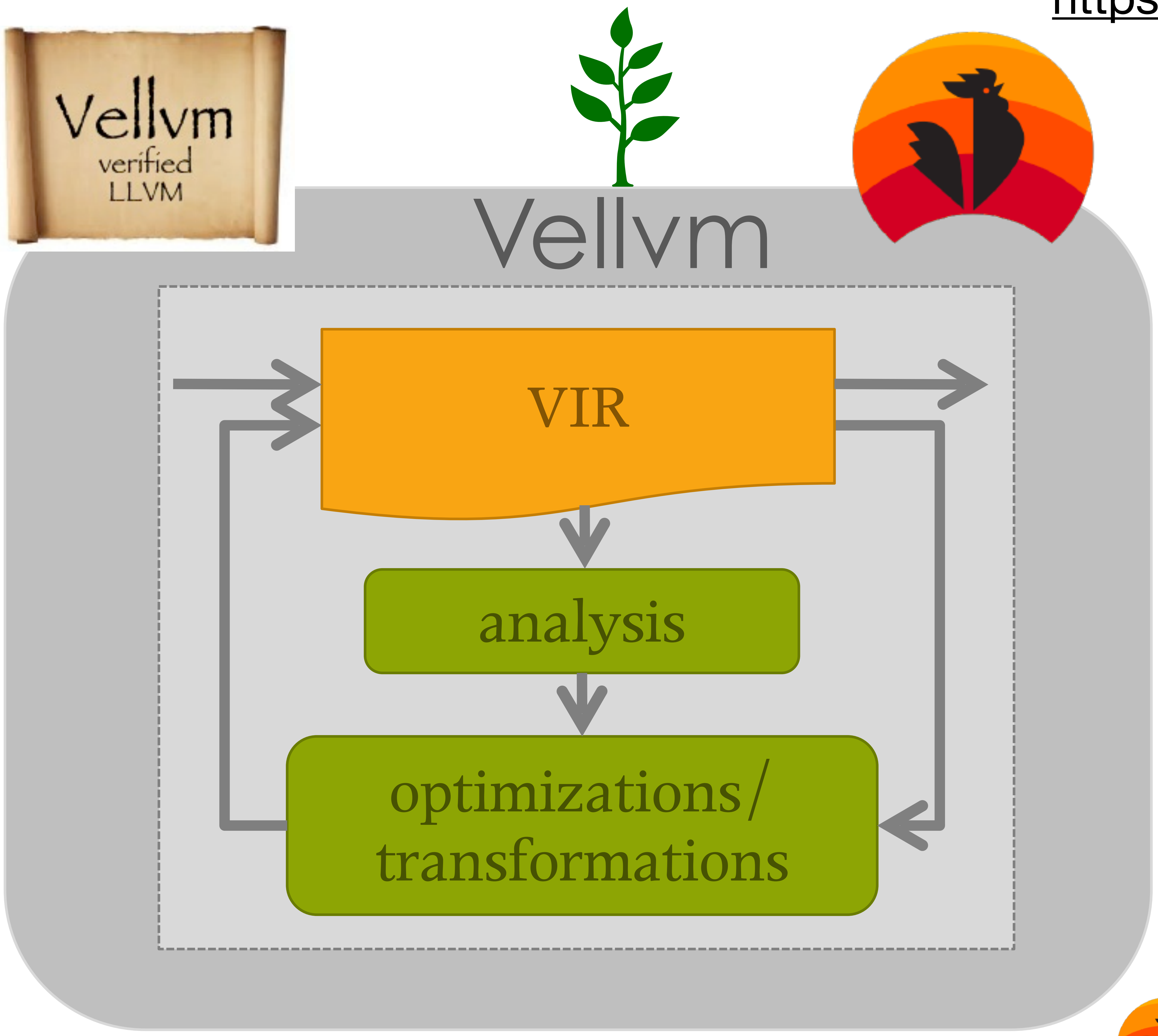
To model C or LLVM, you need to embrace your physicist side!

The Vellvm Project, Revamped

Penn UNIVERSITY of PENNSYLVANIA Inria CMU

https://github.com/vellvm/vellvm

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS



Selected publications and drafts*

[Zakowski et al. - ICFP 2021]
Modular and executable semantics for LLVM IR

[Yoon et al. - ICFP 2022]
Meta-theory for layered monadic interpreters

[Zaliva et al.]
Verified HELIX front-end

[Beck et al.]
Infinite/finite memory model for LLVM IR

[Yoon et al.]
Relational separation logic for LLVM IR

* : all results mechanized in the Coq Proof Assistant

Let's get down to Business

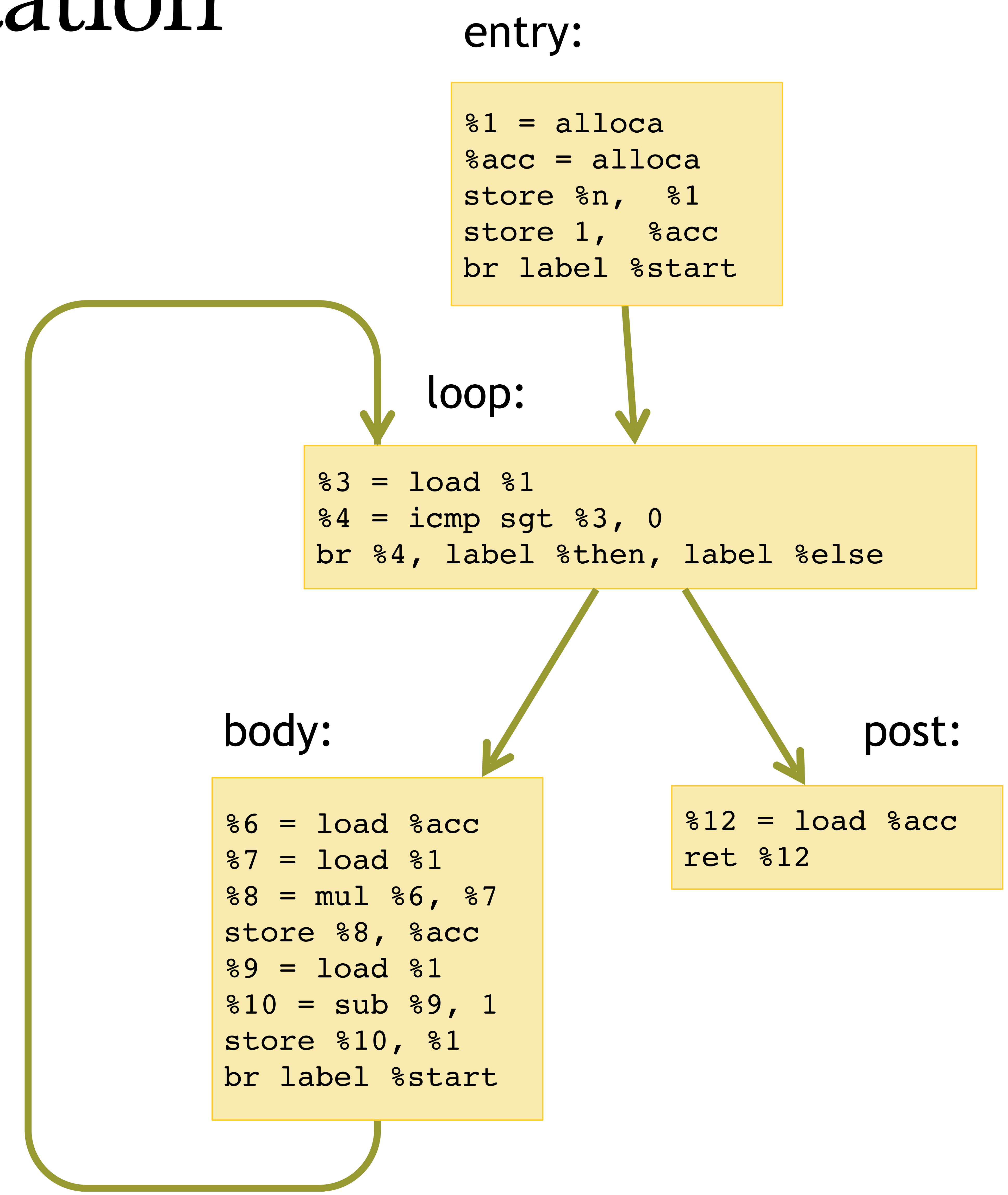
Modular and Executable Semantics for LLVM IR

joint work with

Yannick Zakowski, Calvin Beck, Ilia Zaichuk, Vadim Zaliva, Steve Zdancewic

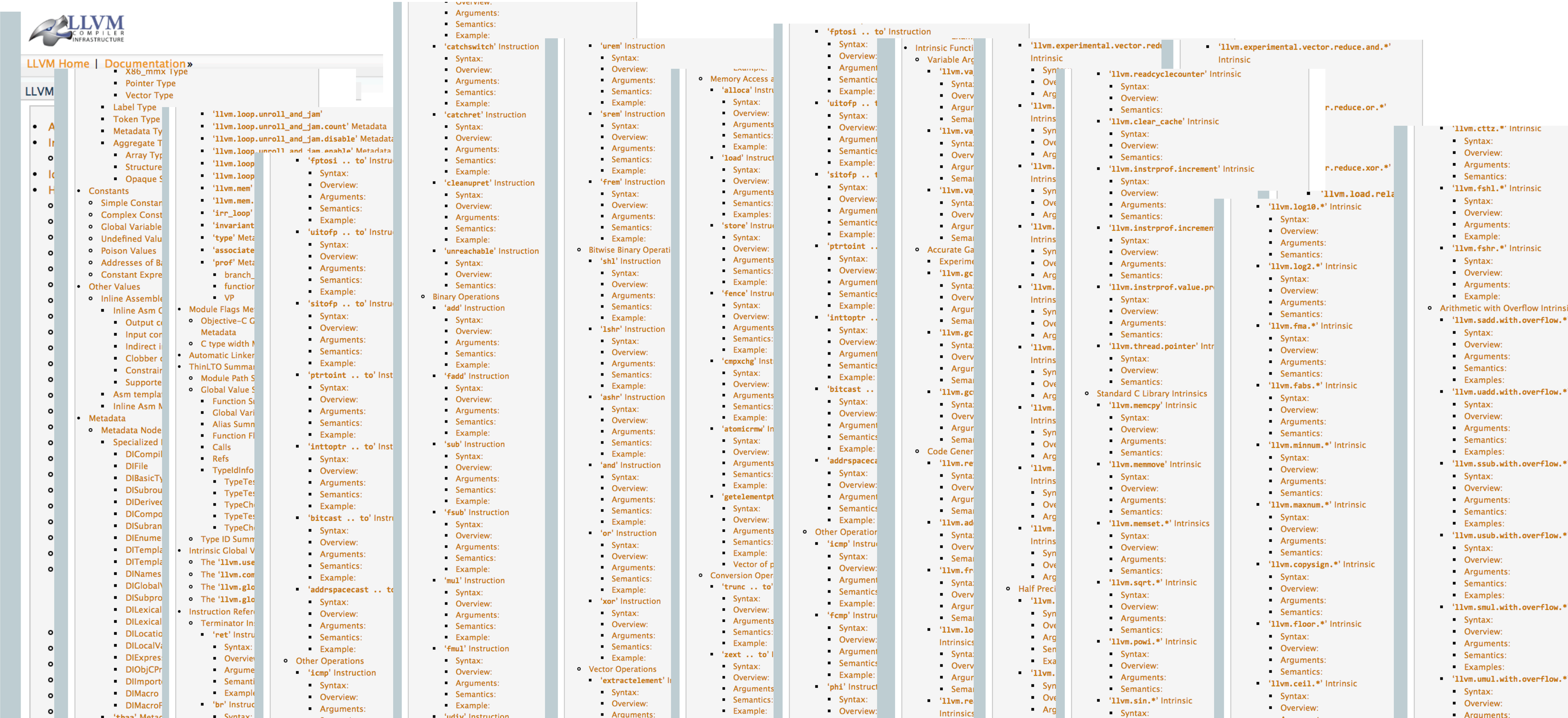
LLVM Intermediate Representation

- LLVM IR
  - Control-flow Graphs:
    - Labeled blocks
    - Straight-line Code
    - Block Terminators
    - Static Single Assignment Form (phi-nodes)
- Types:
  - i64 ⇒ 64-bit integers
  - i64* ⇒ pointer



LLVM LangRef

https://llvm.org/docs/LangRef.html



Studying the (formal) semantics of LLVM IR

Formal Semantics

Crellvm [Kang et al., 18]

K-LLVM [Li and Gunter, 20]

Vellvm [Zhao et al., 12]

Taming UB [Lee et al., 17]

Concurrency [Chakraborty and Vafeiadis, 17]

Realistic Memory Models

Integer-Pointer Cast [Kang et al., 15]

Twin-Allocation [Lee et al., 18]

Bug Finding

Alive [Lopes et al., 15]

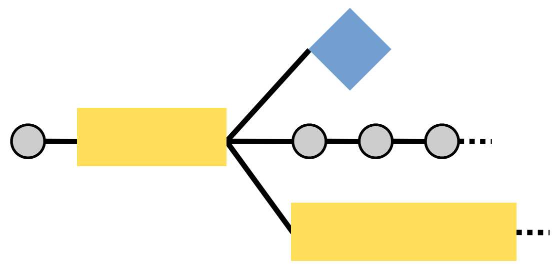
Alive 2 [Lopes et al., 21]

More..

Vellvm 2.0: A redesign of Vellvm

A Coq formal semantics for a large, sequential fragment of LLVM IR coming with:

Interaction Trees (itrees)



[Xia et al. 2020]

github.com/DeepSpec/InteractionTrees

A generic toolkit to define and reason about the semantics of interactive systems

Semantics: Compositional, Modular, Executable

Reasoning: Equational, termination sensitive

Used to build

(Re)Vellvm

github.com/vellvm/vellvm

VIR: a compositional, modular and executable formal semantics for (sequential) LLVM IR

New reasoning principles to verify program transformations

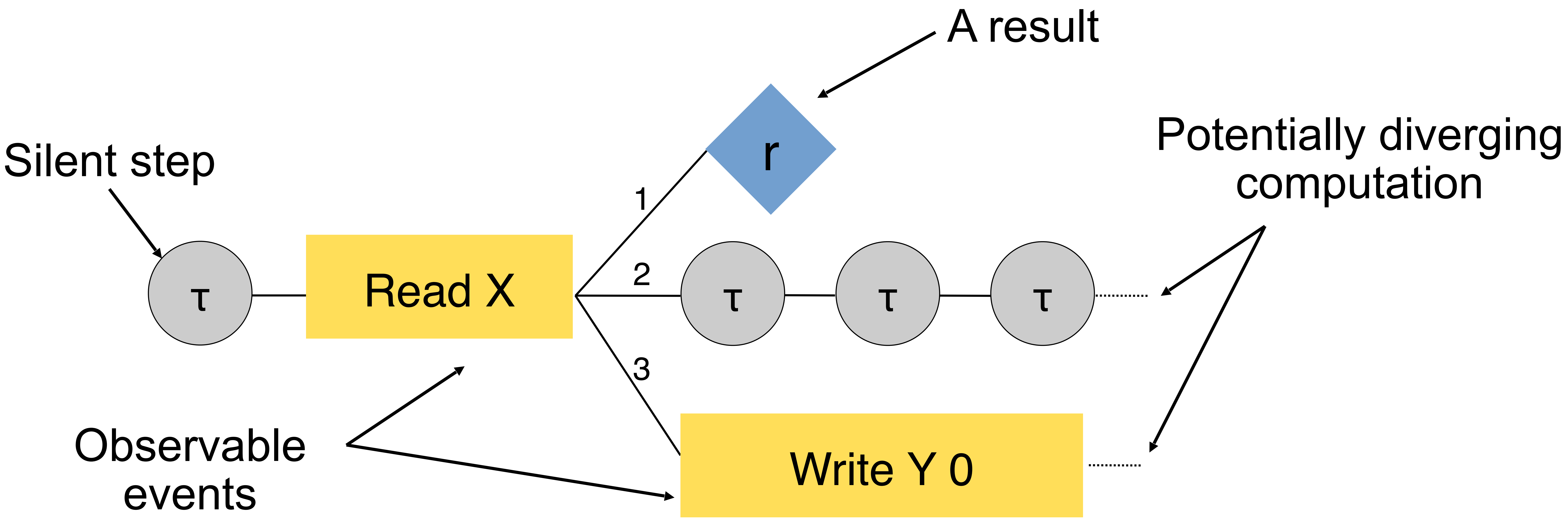
Event-based Semantics

Modularity

Interaction Trees

[Xia et al. 2020]

A data structure for modelling potentially diverging programs in Coq



Interaction Trees

```
CoInductive itree (E: Type -> Type) (R: Type): Type :=
| Ret (r: R)
| Tau (t: itree E R)
| Vis {X: Type} (e: E X) (k: X -> itree E R).
```

A value of the datatype (`itree E R`) represents:

- a potentially diverging computation,
- which may return a value of type R,
- while emitting during its execution events from the interface E.

Interaction Trees

Basic combinators and notation

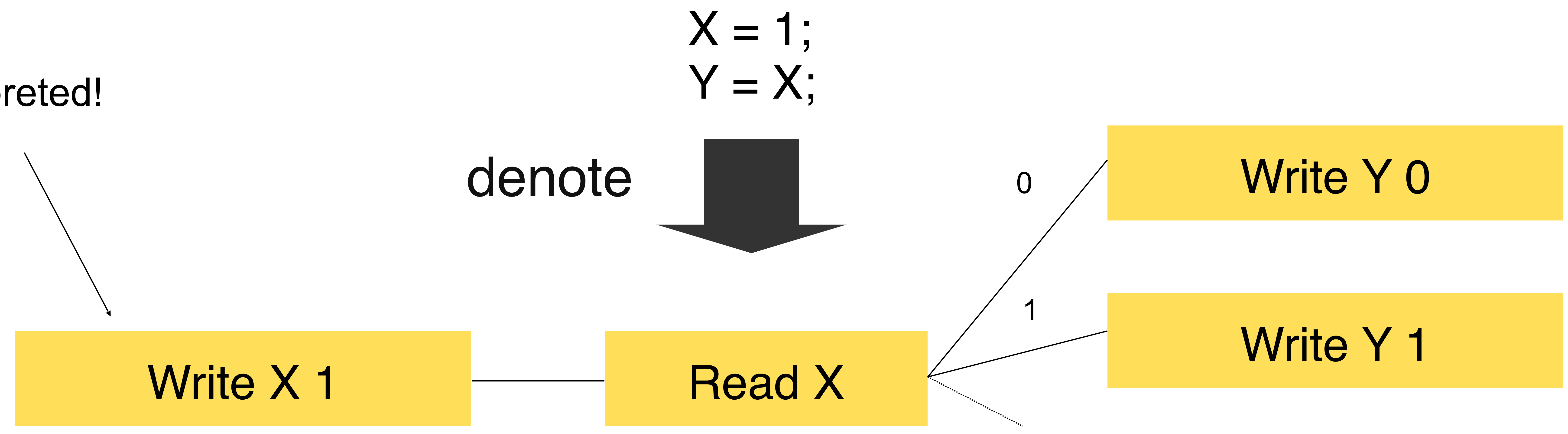
- pure computation: `ret x`
- monadic bind (sequence):
  - `bind t k`
  - `x <- k ;; k x`
- performing an event: `trigger e`
  - note that an "event" is merely syntactic, until it is given a semantics via a handler

Two-phased Denotation

Uninterpreted!

Phase 1

Control flow and potential divergence are internalized

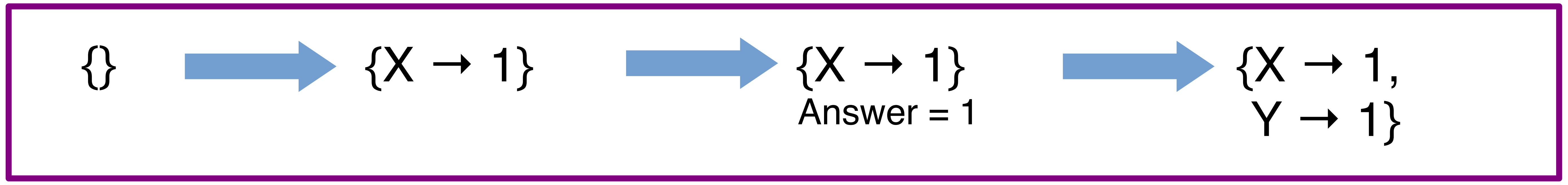


Phase 2

The semantics of effects is introduced

interp handle

The tree is interpreted via an event handler



Defining a modular LLVM IR semantics

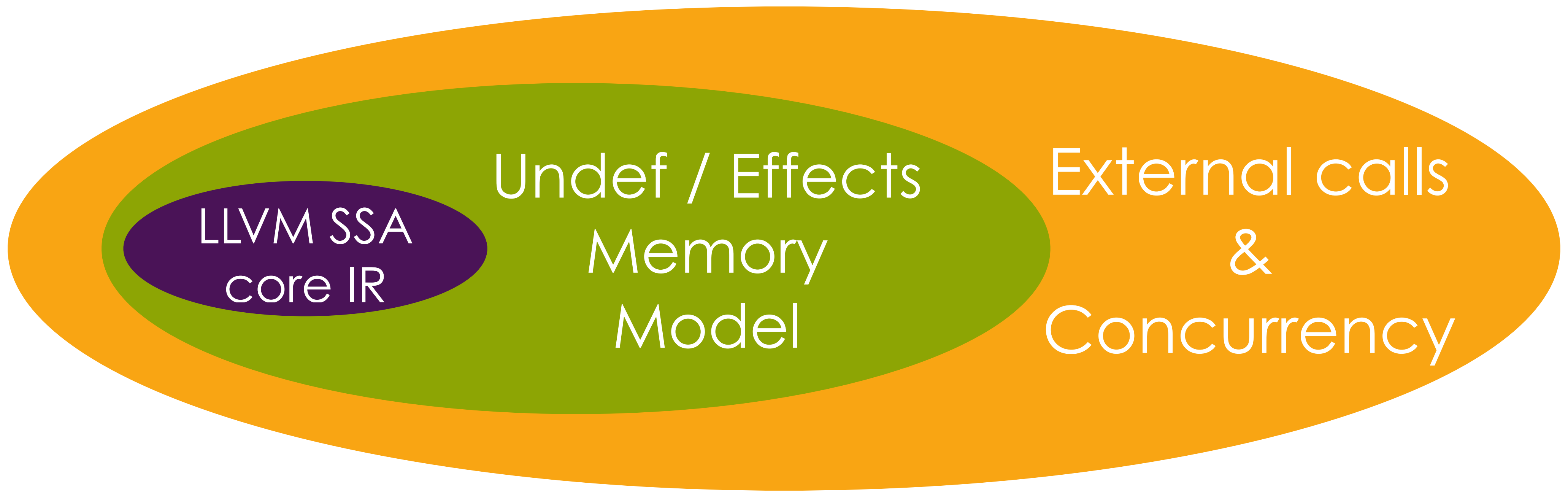
SSA ≈ functional program [Appel 1998]

+

- Undefined values / poison
- Effects
  - structured heap load/store
  - system calls (I/O)
- Types & Memory Layout
  - structured, recursive types
  - type-directed projection

We know how to model this and prove properties about the models.

Q: How do we decompose these effects MODULARLY?



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.

```
(* Interactions with the memory *)
Variant MemoryE : Type → Type :=
| MemPush : MemoryE unit
| MemPop  : MemoryE unit
| Alloca  : ∀ (t:dtyp),                                         (MemoryE dvalue)
| Load    : ∀ (t:dtyp)    (a:dvalue),                           (MemoryE uvalue)
| Store   : ∀ (a:dvalue) (v:dvalue),                            (MemoryE unit)
| GEP     : ∀ (t:dtyp)    (v:dvalue) (vs:list dvalue),          (MemoryE dvalue)
| ItoP    : ∀ (i:dvalue),                                       (MemoryE dvalue)
| PtoI    : ∀ (t:dtyp) (a:dvalue),                              (MemoryE dvalue)
.
```

Interpreting LLVM Events

1. Denoting events

```
(* Interactions with the memory *)
Variant MemoryE : Type → Type :=
| MemPush : MemoryE unit
| MemPop  : MemoryE unit
| Alloca  : ∀ (t:dtyp),                                  (MemoryE dvalue)
| Load    : ∀ (t:dtyp)   (a:dvalue),                     (MemoryE uvalue)
| Store   : ∀ (a:dvalue) (v:dvalue),                     (MemoryE unit)
| GEP     : ∀ (t:dtyp)   (v:dvalue) (vs:list dvalue),    (MemoryE dvalue)
| ItoP    : ∀ (i:dvalue),                                (MemoryE dvalue)
| PtoI    : ∀ (t:dtyp) (a:dvalue),                       (MemoryE dvalue)
.
```

2. Giving semantics to events

Generalized semantic domain: the resulting events contain failure and undefined behavior events

```
Definition handle_memory {E} `{FailureE -< E} `{UBE -< E}: MemoryE ~> stateT memory_stack (itree E)
```

3. Fold and layer

```
Definition interp {E M : Type -> Type} (h : E ~> M) : itree E ~> M
```

Fold and layer?

3. Fold and layer

```
Definition interp {E M : Type -> Type} (h : E ~> M) : itree E ~> M
```

An event handler [h] defines a monad morphism (i.e. respects bind and ret*) for any monad [M] with a loop operator.

"fold over a tree, and return something you can iterate over"

[Yoon et al. - ICFP 2022]
Meta-theory for layered monadic interpreters

*and additionally, rules for iteration

Layered interpretation, plug-and-play

	`VIR`
	↓ *structural representation*
Level 0	`itree VellvmE` $\mathcal{V}_u$
	↓ ***intrinsics***
Level 1	`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 `)` $\mathcal{V}_u$

Layered interpretation, plug-and-play

`VIR`

↓ *structural representation*

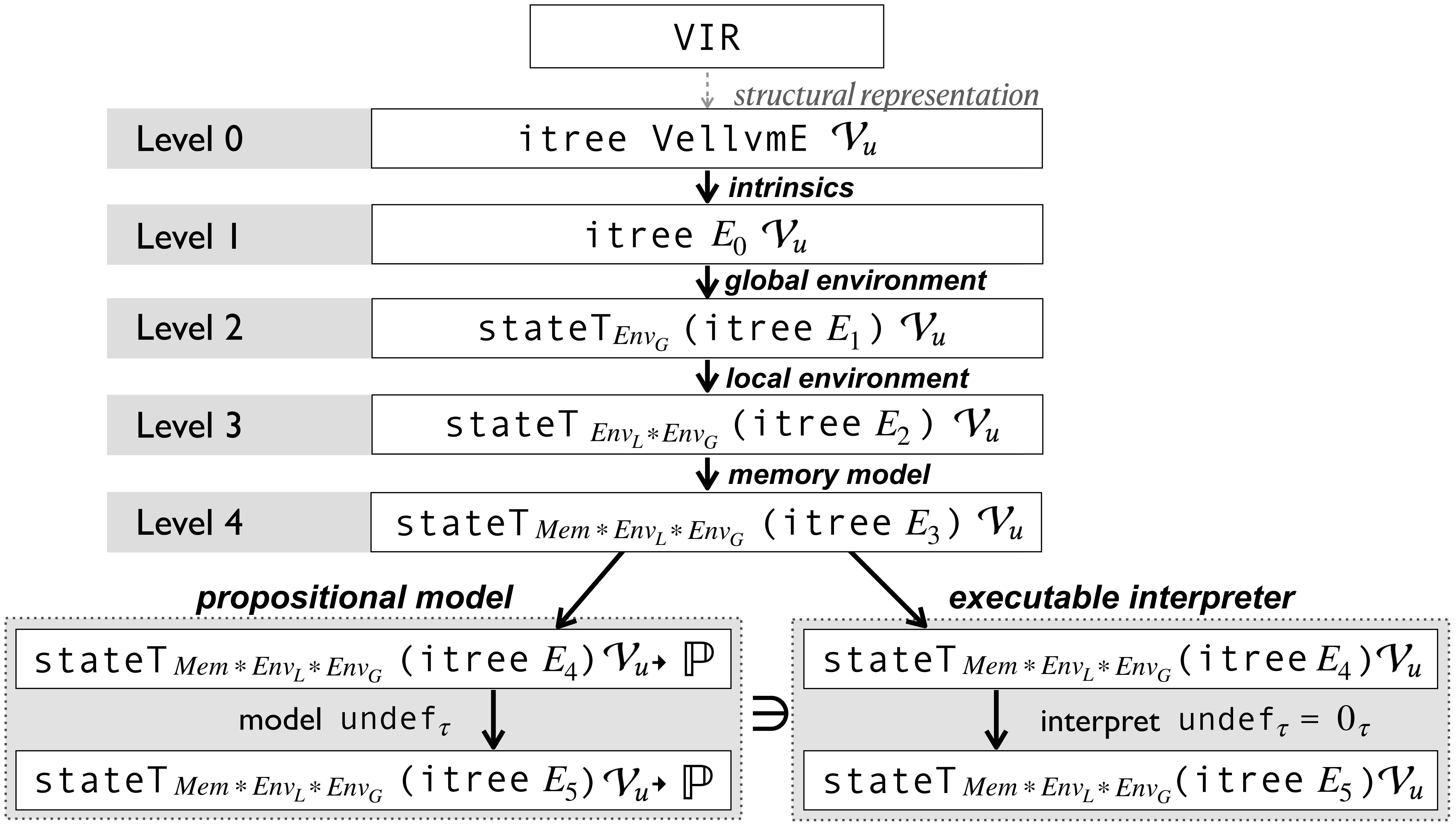
Level 0	`itree VellvmE` $\mathcal{V}_u$
	↓ ***intrinsics***
Level 1	`itree` E_0 $\mathcal{V}_u$
	↓ ***global environment***
Level 2	`stateT`$_{Env_G}$ `(itree` E_1`)` $\mathcal{V}_u$
	↓ ***local environment***
Level 3	`stateT`$_{Env_L * Env_G}$ `(itree` E_2`)` $\mathcal{V}_u$
	↓ ***memory model***
Level 4	`stateT`$_{Mem * Env_L * Env_G}$ `(itree` E_3`)` $\mathcal{V}_u$

Is this the right memory model?

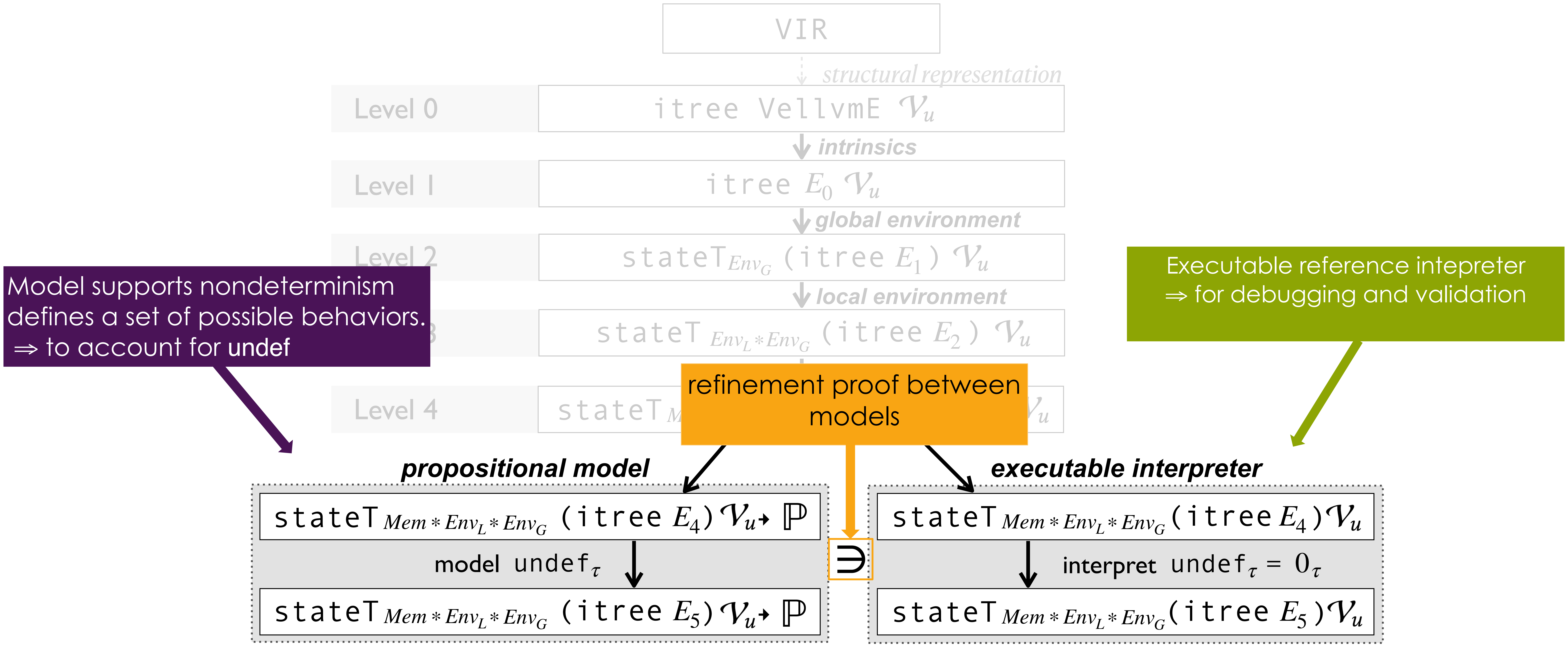
[Beck et al.]

Infinite/finite memory model for LLVM IR

Layered interpretation, plug-and-play



Layered interpretation, plug-and-play



Denotational Semantics

Compositionality

Denotational semantics for LLVM IR

$\llbracket - \rrbracket_{\text{expr}}$ expressions

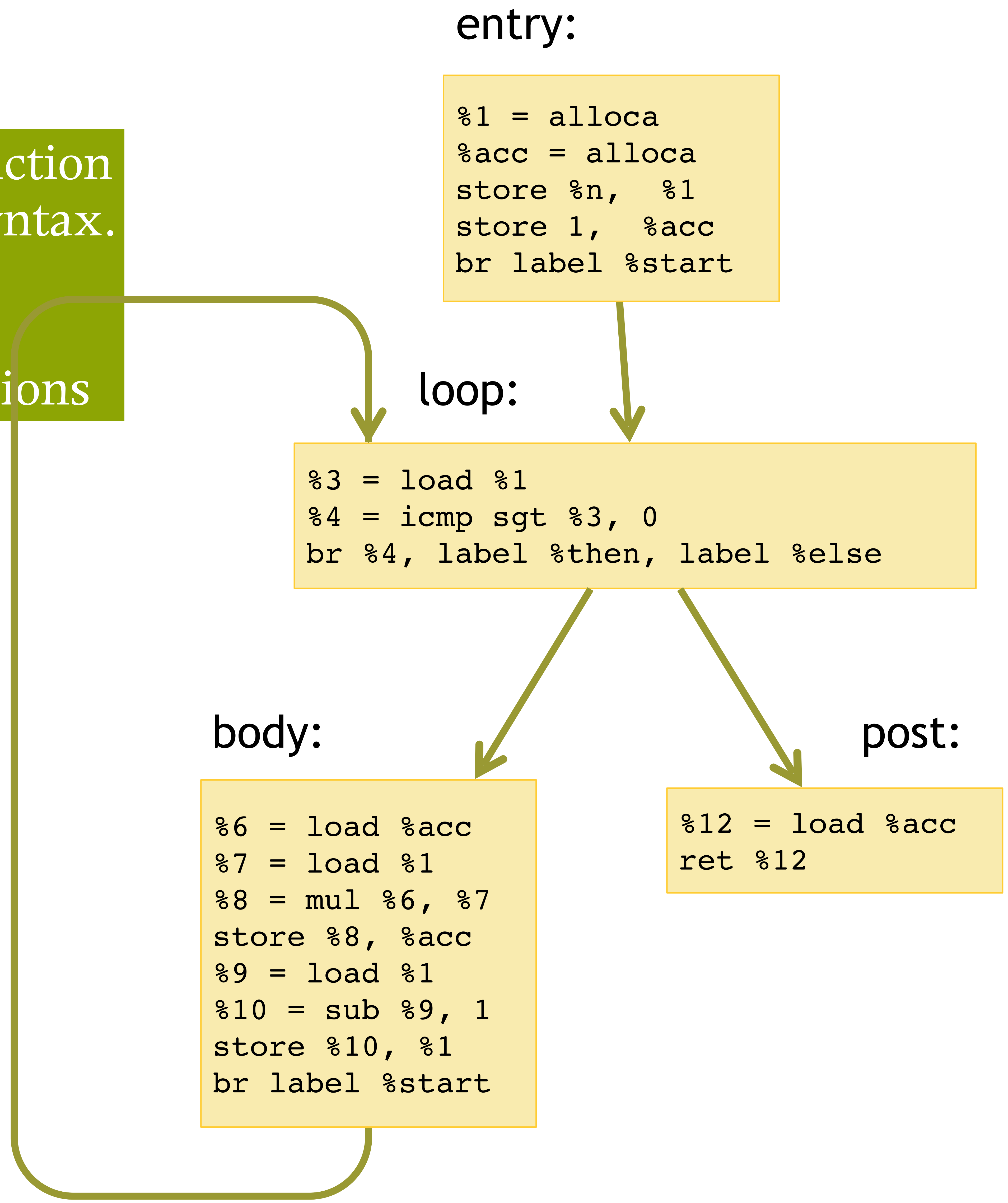
$\llbracket - \rrbracket_{\text{instr}}$ instructions

$\llbracket - \rrbracket_{\text{block}}$ blocks

$\llbracket - \rrbracket_{\text{cfg}}$ control-flow graphs

$\llbracket - \rrbracket_{\text{llvm}}$ programs
(mutually recursive functions)

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



Denoting expressions

"Simple arithmetic"

- Division by zero
  - Undefined behavior: behavior undefined by the language standard ("no promises!")
  - Compilers often assume source has no UB
- undef: set of defined values at a certain type
  - x + x ≠ 2 * x
- poison: "deferred undefined behavior", a value useful for representing signed overflow in LLVM IR

$$\begin{aligned}\llbracket e_1/e_2 \rrbracket_e = {} & uv_1 \leftarrow \llbracket e_1 \rrbracket_e \;;; uv_2 \leftarrow \llbracket e_2 \rrbracket_e \;;; \\ & dv \leftarrow \mathtt{pick}(uv_2) \;;; uv_1 \oslash dv\end{aligned}$$

General recursion in Gallina

Gallina

pure fragment of OCaml, with dependent types

```
Fixpoint fact n :=
  match n with
  | 0   => 1
  | S m => n * fact m
```

Ok!

```
Fixpoint zip xs ys :=
  match xs with
  | nil   => ys
  | x::xs => x::zip ys xs
```

Prove it!

```
Fixpoint interp c s :=
 match c with
 | ..
 | while b c =>
   if is_true b s
   then interp (while b c) (interp c s)
   else s
```

Tough luck!

Combinator for recursion

First step : modeling tail-recursive calls

- Iteration as a primitive for tail recursion
- Iterate a function updating an accumulator [A], until it produces an output [B]

```
iter : (A → itree E (A + B)) → (A → itree E B)
```

```
CoFixpoint iter (body : A → itree E (A + B))
  : A → itree E B :=
  fun a ⇒ ab ← body a ;;
          match ab with
          | inl a ⇒ Tau (iter body a)
          | inr b ⇒ Ret b
          end.
```

Iterative laws

```
         iter f  ≈  f >>> case_ (iter f) id_
   iter f >>> g  ≈  iter (f >>> bimap id_ g)
```

Mutual recursion combinator

- Technique for general recursion developed by McBride 2015
- Signature of a mutually recursive function

```
Inductive ackermannE : Type → Type :=
| Ackermann : nat → nat → ackermannE nat.
```

- Body of function makes recursive calls with 'Ackermann' events without ensuring well-foundedness

```
Definition h_ackermann : ackermannE ⇝ itree (ackermannE +' emptyE) :=
  fun _ '(Ackermann m n) ⇒ if m =? 0 then Ret (n + 1)
                            else if n =? 0 then trigger (inl1 (Ackermann (m-1) 1))
                            else (ack ← trigger (inl1 (Ackermann m (n-1))) ;;
                                  trigger (inl1 (Ackermann (m-1) ack))).
```

- Given the recursive handler and [mrec] combinator, we can tie the knot

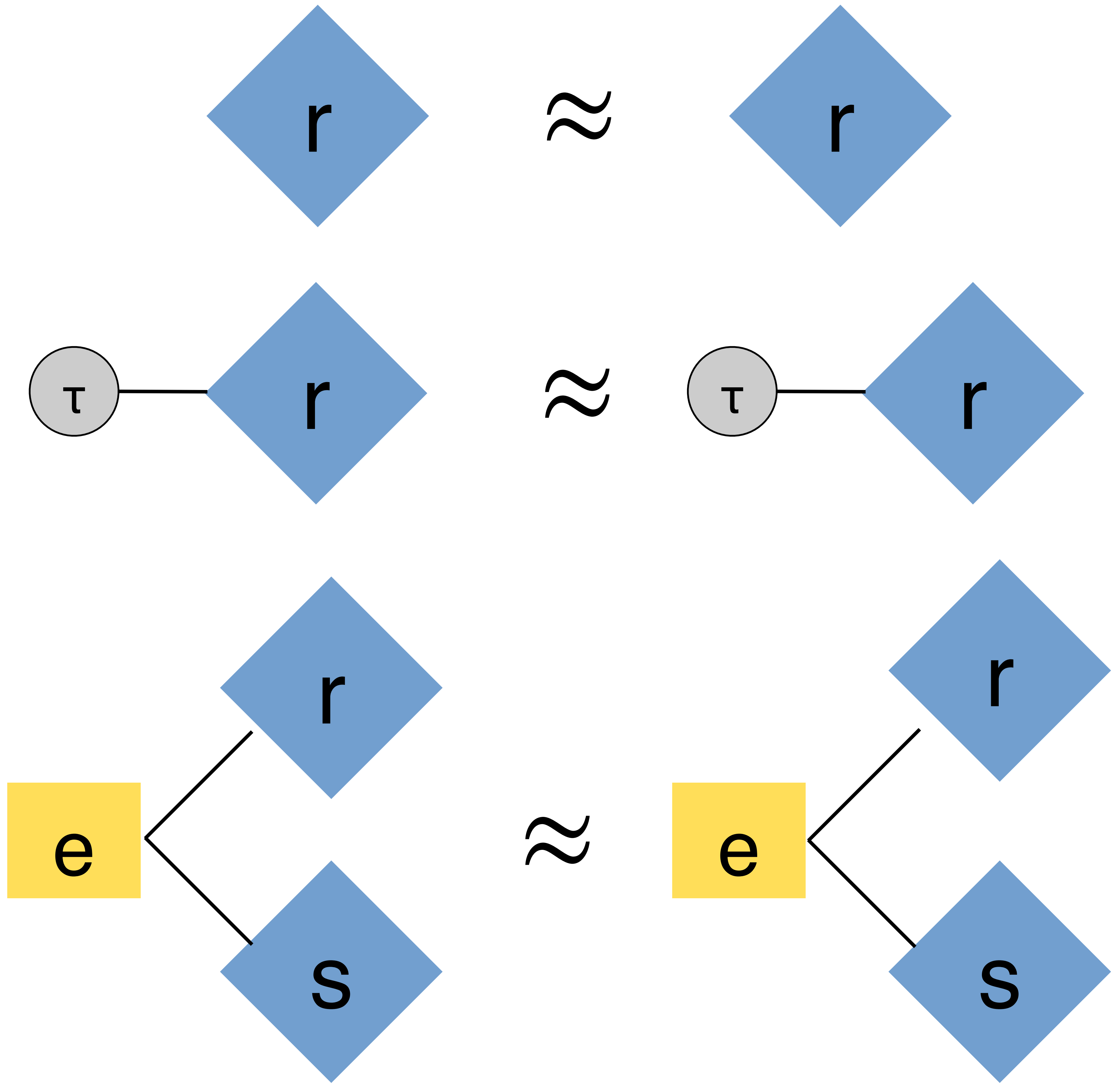
```
Definition ackermann : nat → nat → itree emptyE nat :=
  fun m n ⇒ mrec h_ackermann (Ackermann m n).
```

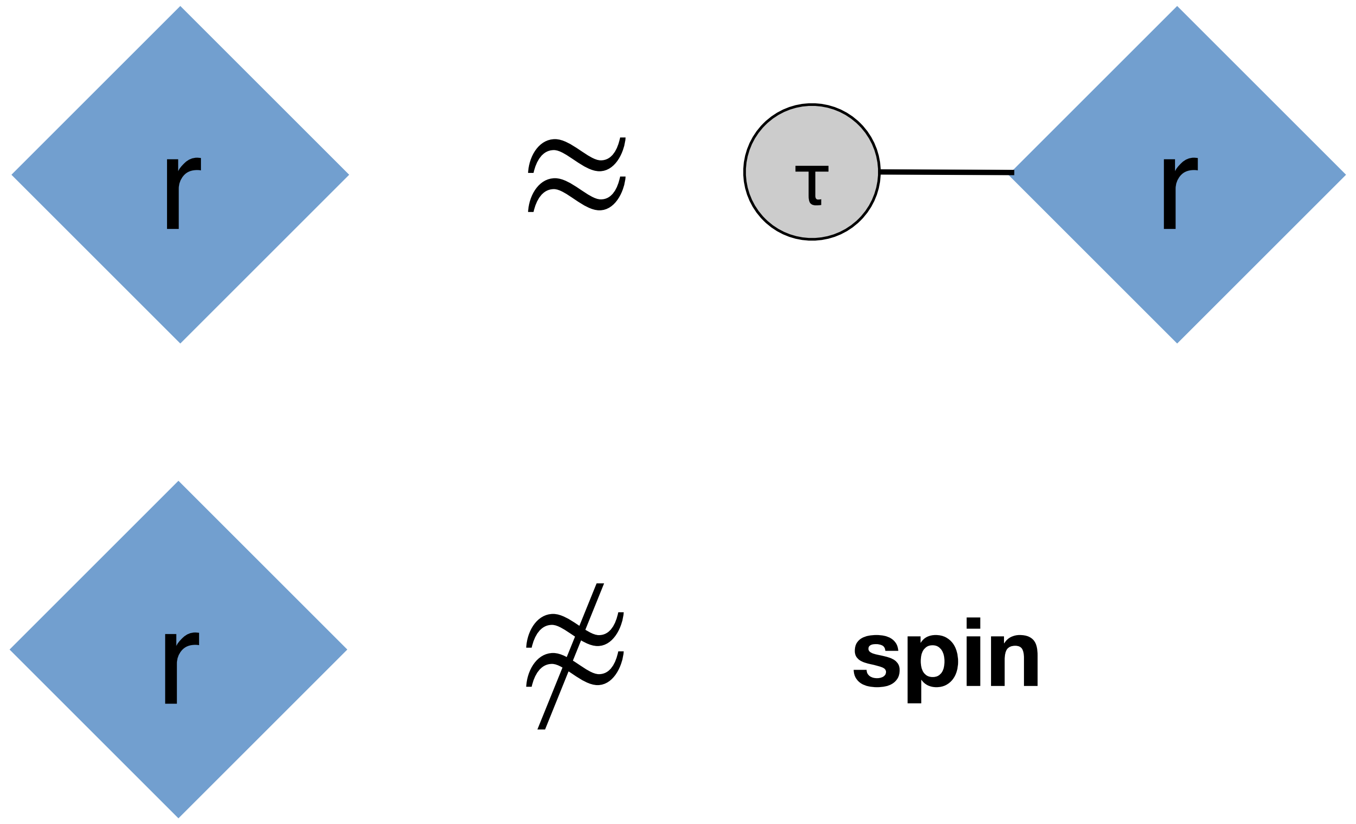
Weak Bisimulation

gpaco [Zakowski et al. 2020]

(Coinductive) relation ignoring finite amount of internal steps

eutt: “equivalence up-to taus”





Interaction Tree equational theory (excerpt)

Structural laws

```
                (Tau t) ≈ t
     (x ← (Tau t) ;; k) ≈ Tau (x ← t ;; k)
(x ← (Vis e k1) ;; k2) ≈
            (Vis e (fun y ⇒ (k1 y) ;; k2))
```

Monad laws

```
    (x ← ret v ;; k x) ≅ (k v)
      (x ← t ;; ret x) ≅ t
(x ← (y ← s ;; t) ;; u) ≅ (y ← s ;; x ← t ;; u)
```

Iterative laws

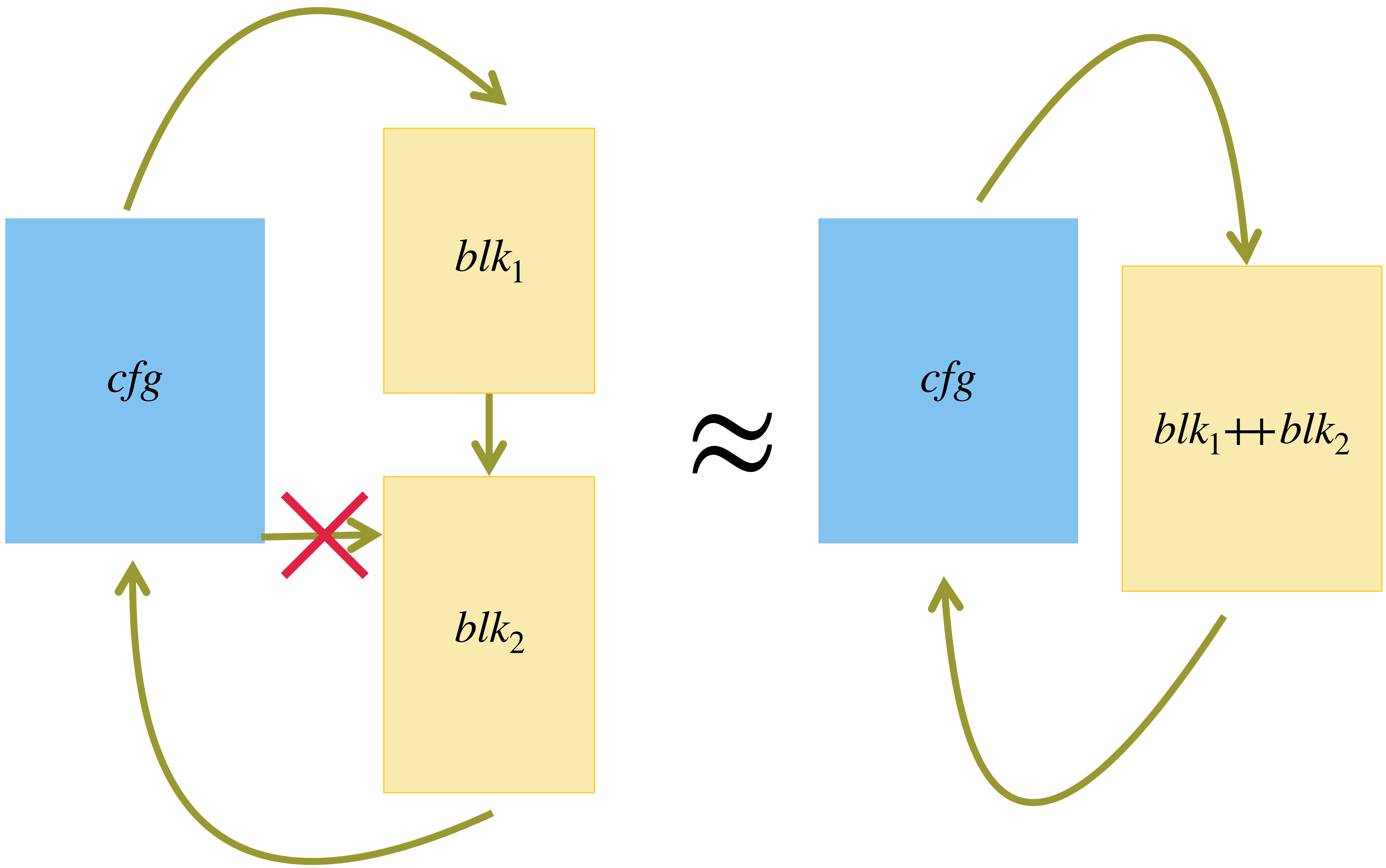
```
     iter f   ≈̂   f >>> case_ (iter f) id_
iter f >>> g  ≈̂   iter (f >>> bimap id_ g)
```

Interp laws

```
interp h (trigger e)    ≅ h _ e
interp h (Ret r)        ≅ ret r
interp h (x ← t;; k x) ≅
   x ← (interp h t);; interp h (k x)
```

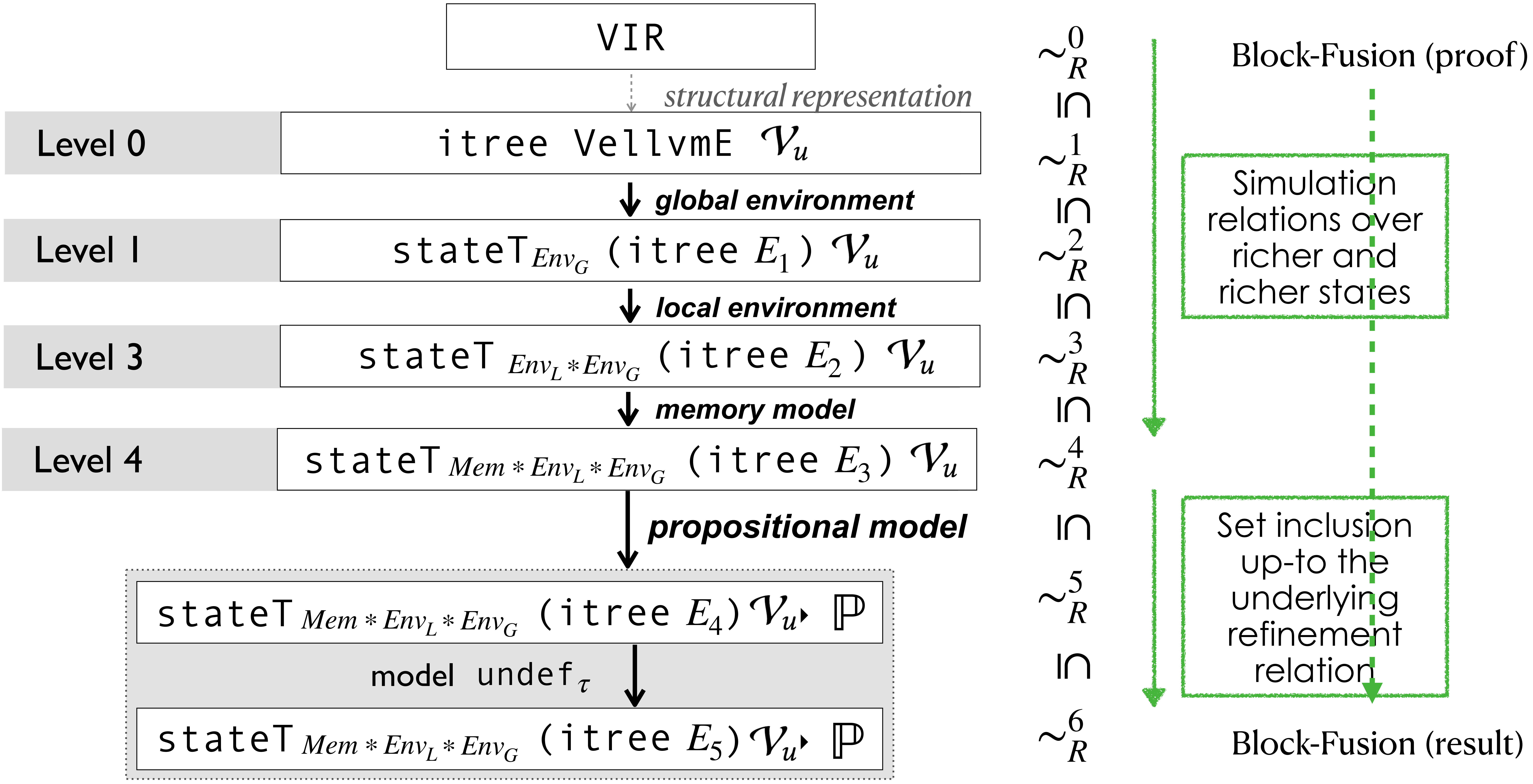
Benefits of Interaction-Tree based reasoning

Reasoning about control-flow



- Proof of block-merging optimization
- Reasoning about composing control-flow operators is simple
- Benefit

  Proof involves reasoning only about control flow, not other side-effects (e.g. state, exception..)



VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
global environment
Level 1
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
stateT $_{Mem * Env_L * Env_G}$ (itree E_4) $\mathcal{V}_u$ $\mathbb{P}$
model undef$_\tau$
stateT $_{Mem * Env_L * Env_G}$ (itree E_5) $\mathcal{V}_u$ $\mathbb{P}$
$\sim_R^0 \supseteq \sim_R^1 \supseteq \sim_R^2 \supseteq \sim_R^3 \supseteq \sim_R^4 \supseteq \sim_R^5 \supseteq \sim_R^6$
Block-Fusion (proof)
Simulation relations over richer and richer states
Set inclusion up-to the underlying refinement relation
Block-Fusion (result)

The need for a state-aware program logic

Stateful reasoning in VIR

- Relational reasoning on ITree-based semantics

$$e_t \approx_R e_s$$

Two programs e_t and e_s

(1) Both terminate and satisfy the postcondition R over the result of the computation, OR

(2) Both diverge in simulation with each other

- Stateful Hoare-style reasoning

Given a stateful interpretation function $[\![-]\!]$: `itree (E +' F) A → stateT S (itree F) A`

$$\{\mathscr{P}\}e_t \approx e_s\{\mathscr{Q}\} := \forall \sigma_t, \sigma_s \,.\, \mathscr{P}(\sigma_t, \sigma_s) \Rightarrow [\![e_t]\!] \sigma_t \approx_{\mathscr{Q}} [\![e_s]\!] \sigma_s$$

- Localize reasoning about state using <u>separation logic</u>

Coq Extraction

Executability

Reference Interpreter: Executability

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

start:
  %2 = load i64, i64* %1
  %3 = icmp sgt i64 %2, 0
  br i1 %3, label %then, label %end

then:
  %4 = load i64, i64* %acc
  %5 = load i64, i64* %1
  %6 = mul i64 %4, %5
  store i64 %6, i64* %acc
  %7 = load i64, i64* %1
  %8 = sub i64 %7, 1
  store i64 %8, i64* %1
  br label %start

end:
  %9 = load i64, i64* %acc
  ret i64 %9
}

define i64 @main(i64 %argc, i8** %arcv) {
  %1 = alloca i64
  store i64 0, i64* %1
  %2 = call i64 @factorial(i64 5)
  ret i64 %2
}
```

→ Parser → Extracted interpreter → Tiny OCaml driver to crawl the tree → 120

External calls
Debugging messages
Failure

Tested against clang over:

- A collection of unit tests
- A handful of significant programs from the HELIX frontend
- Experiments over randomly generated programs using QuickChick

Reference Interpreter Validation : Executability

- Debugging

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

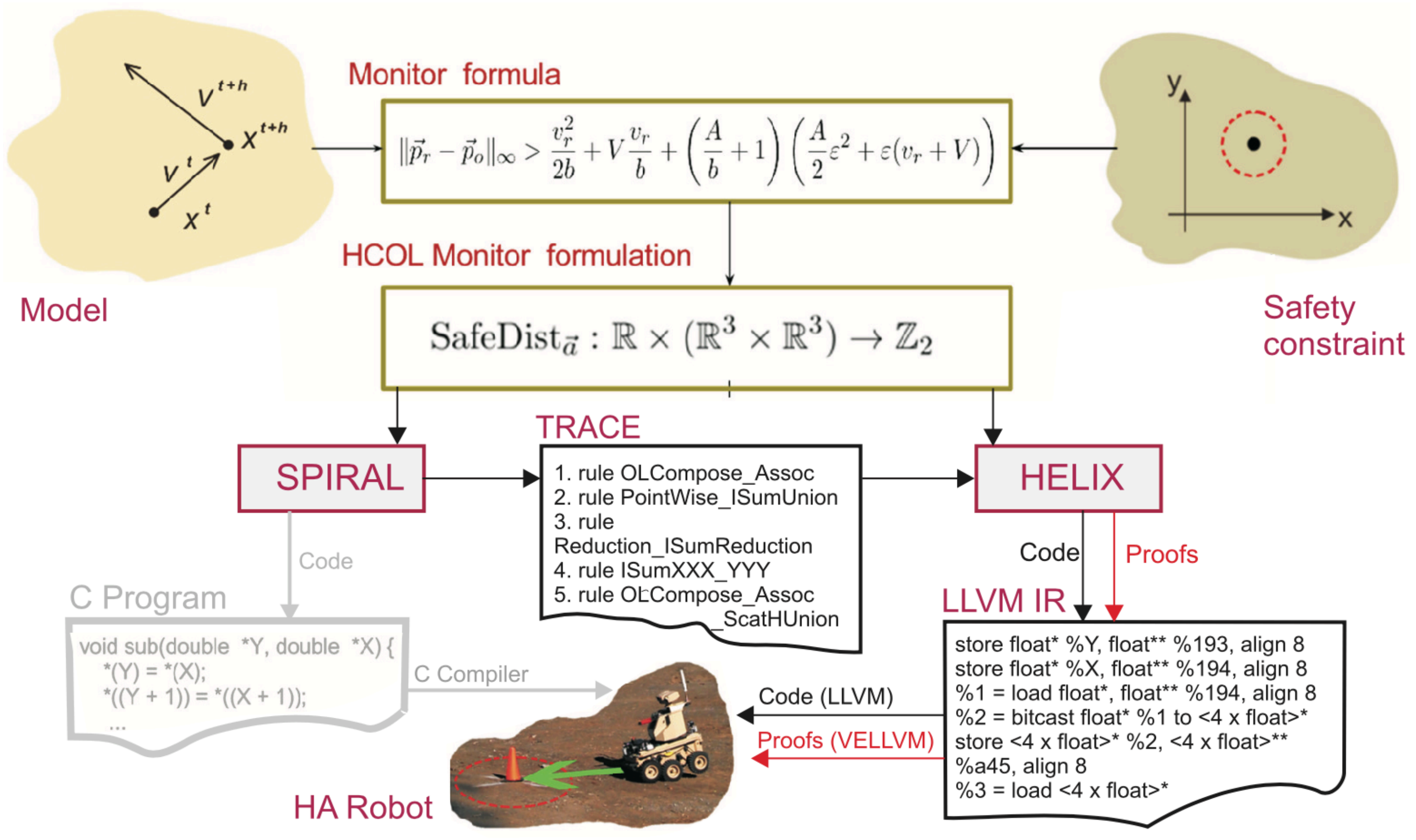
- Find bugs in the existing LLVM infrastructure
  - thinking hard about corner cases while formalizing is a good way to find real bugs
  - identify inconsistent assumptions about the LLVM compiler

- Property-based testing: QuickChick-based generator to generate interesting, UB-free LLVM IR programs (in-progress work, Chen et al.)

SPIRAL/HELIX

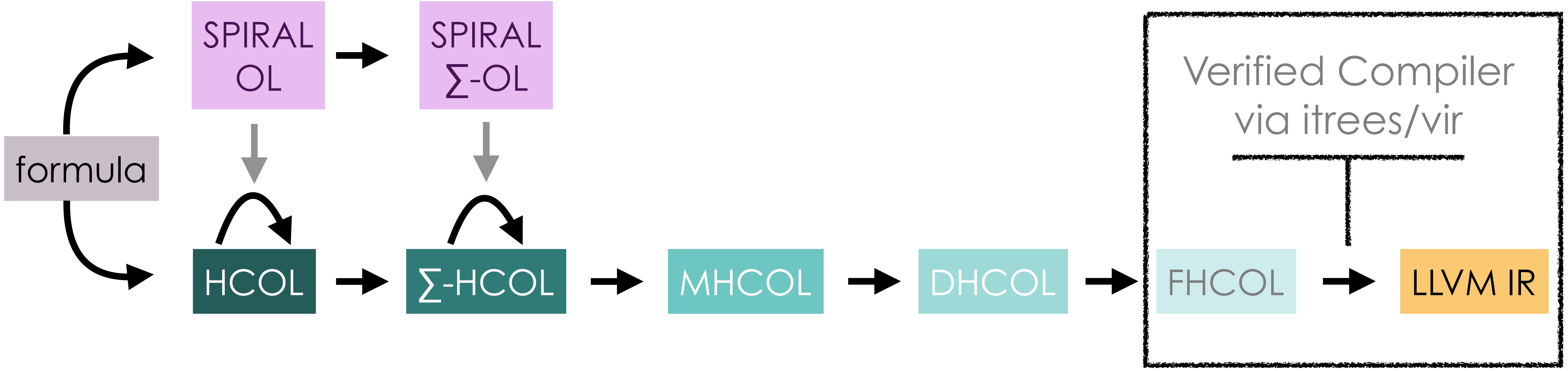
[Püschel, et al. 2005] [Franchetti et al., 2005, 2018] [Zaliva et al., 2015 2018, 2019]

DSL for high-performance numerical computing.



- Numerical computations compiled down to LLVM IR
- Formalized in Coq, targets Vellvm
- Bottom of the compilation chain proved* w.r.t. this technique

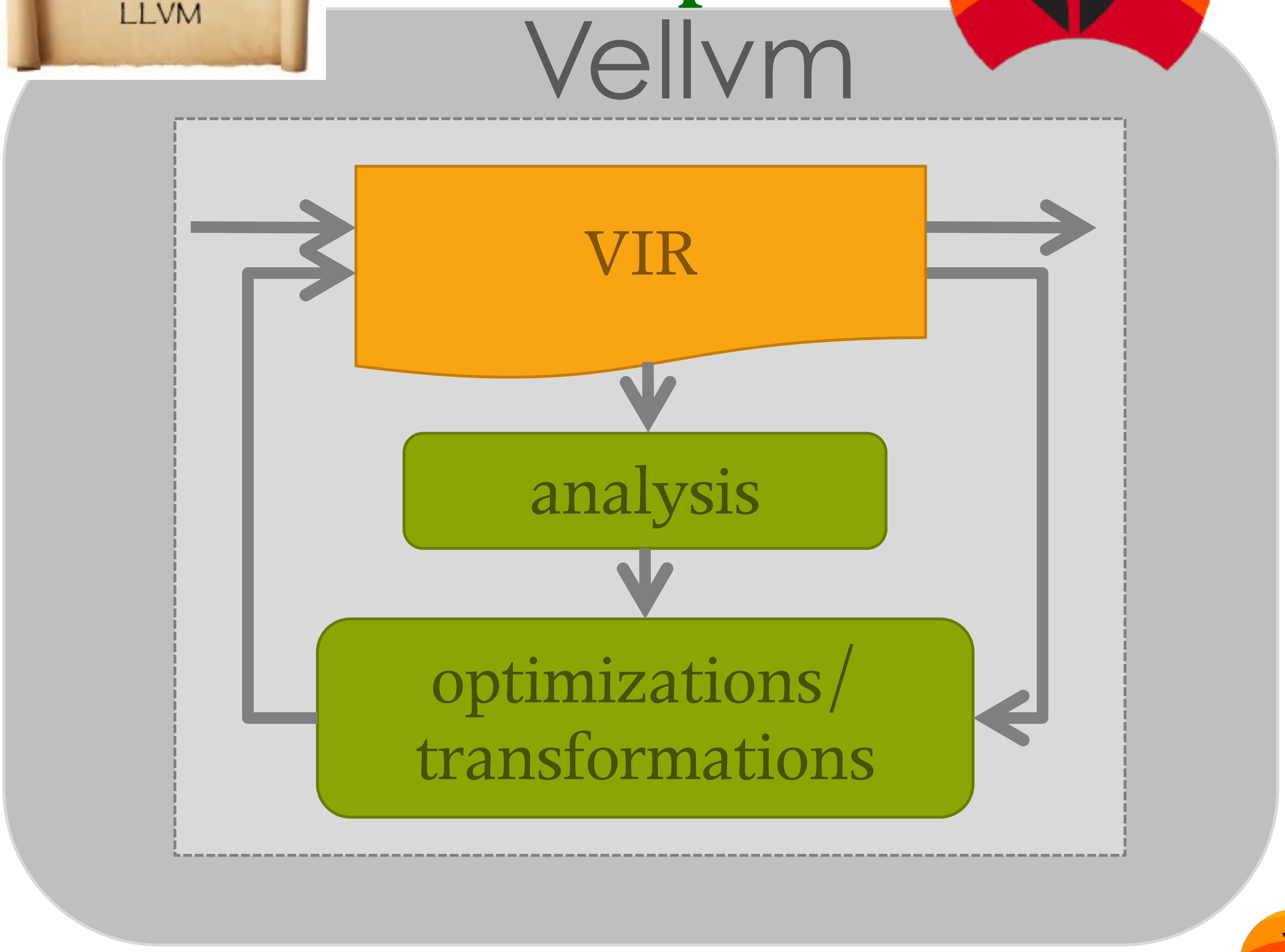
* Some operators are currently not proved



The Vellvm Project, Revamped

Penn University of Pennsylvania · Inria · CMU · MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

https://github.com/vellvm/vellvm



Selected publications and drafts*

[Zakowski et al. - ICFP 2021]
Modular and executable semantics for LLVM IR

[Yoon et al. - ICFP 2022]
Meta-theory for layered monadic interpreters

[Zaliva et al.]
Verified HELIX front-end

[Beck et al.]
Infinite/finite memory model for LLVM IR

[Yoon et al.]
Relational separation logic for LLVM IR

* : all results mechanized in the Coq Proof Assistant