Modular, Compositional, and Executable Formal Semantics for LLVM IR

ICFP 2021

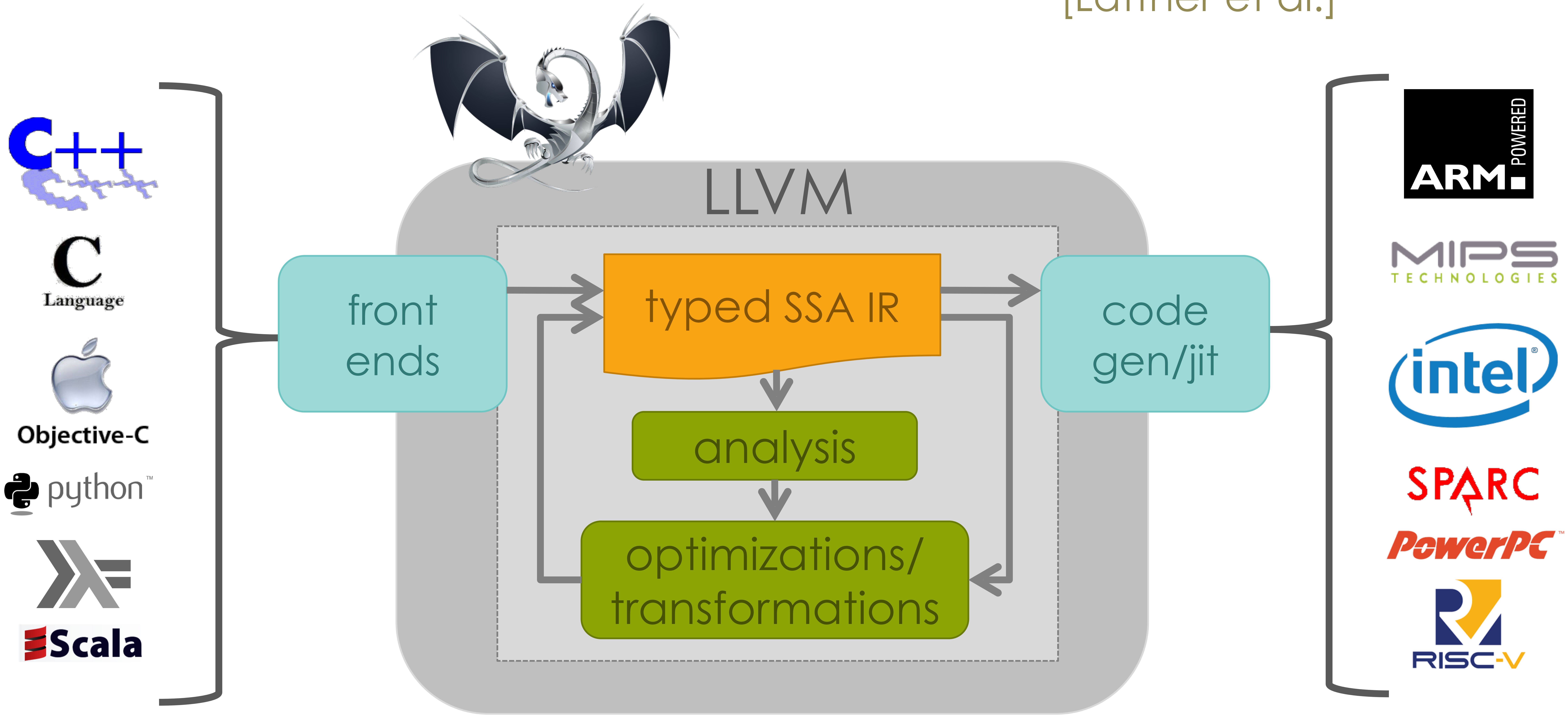
Yannick Zakowski
Calvin Beck
Irene Yoon
Ilia Zaichuk
Vadim Zaliva
Steve Zdancewic

Inria

Penn
University of Pennsylvania

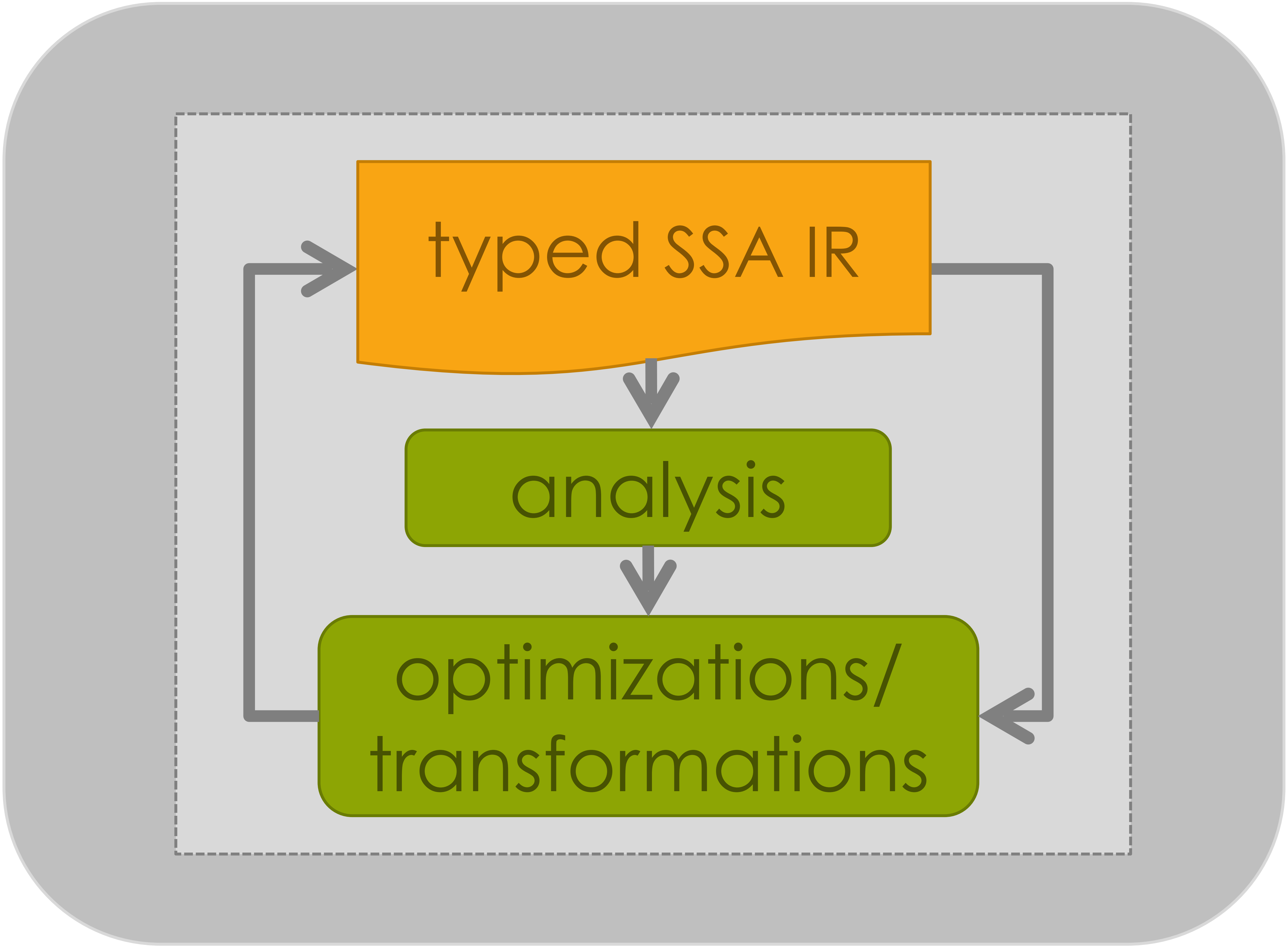
LLVM Compiler Infrastructure

[Lattner et al.]

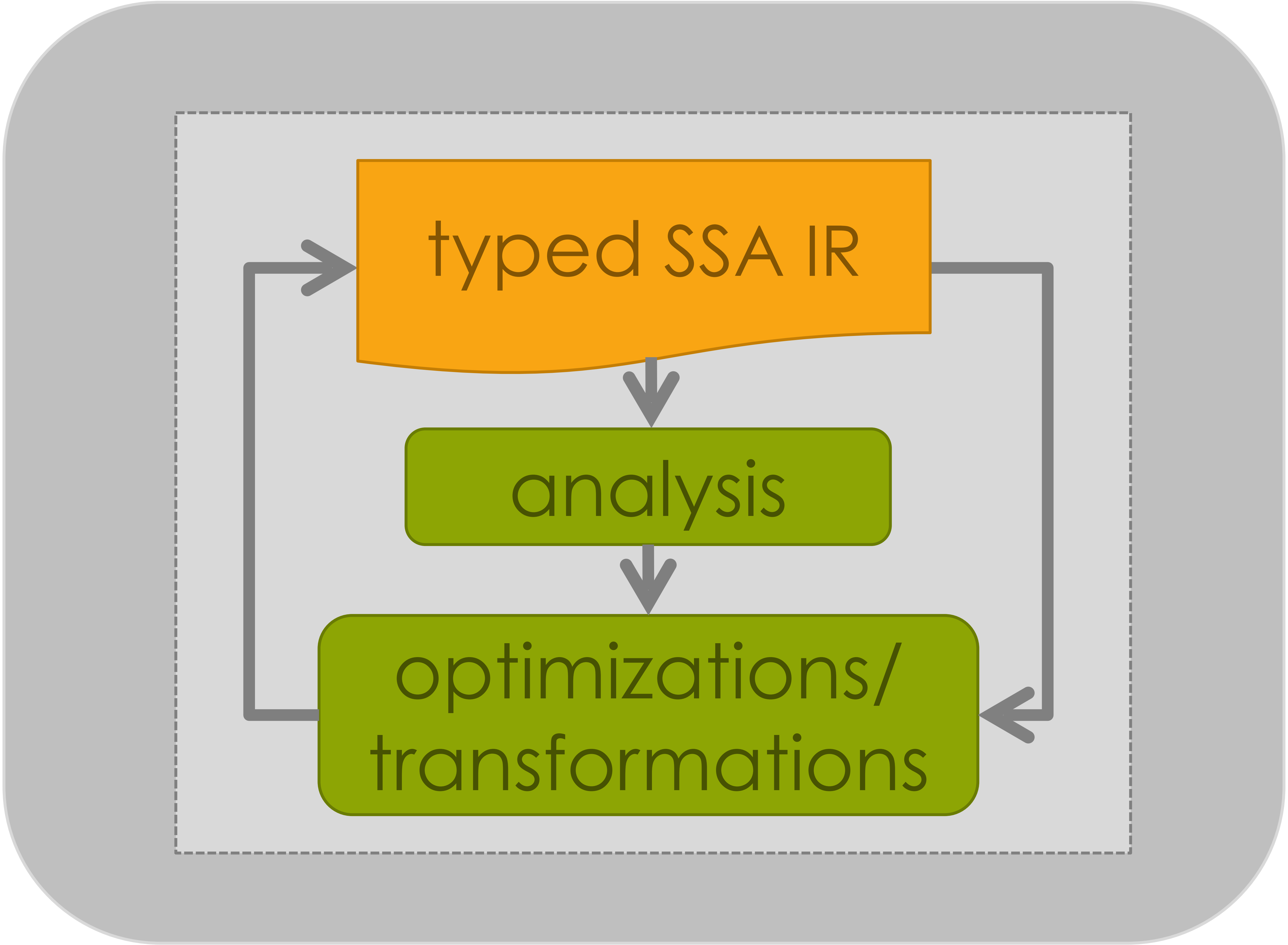


LLVM Compiler Infrastructure

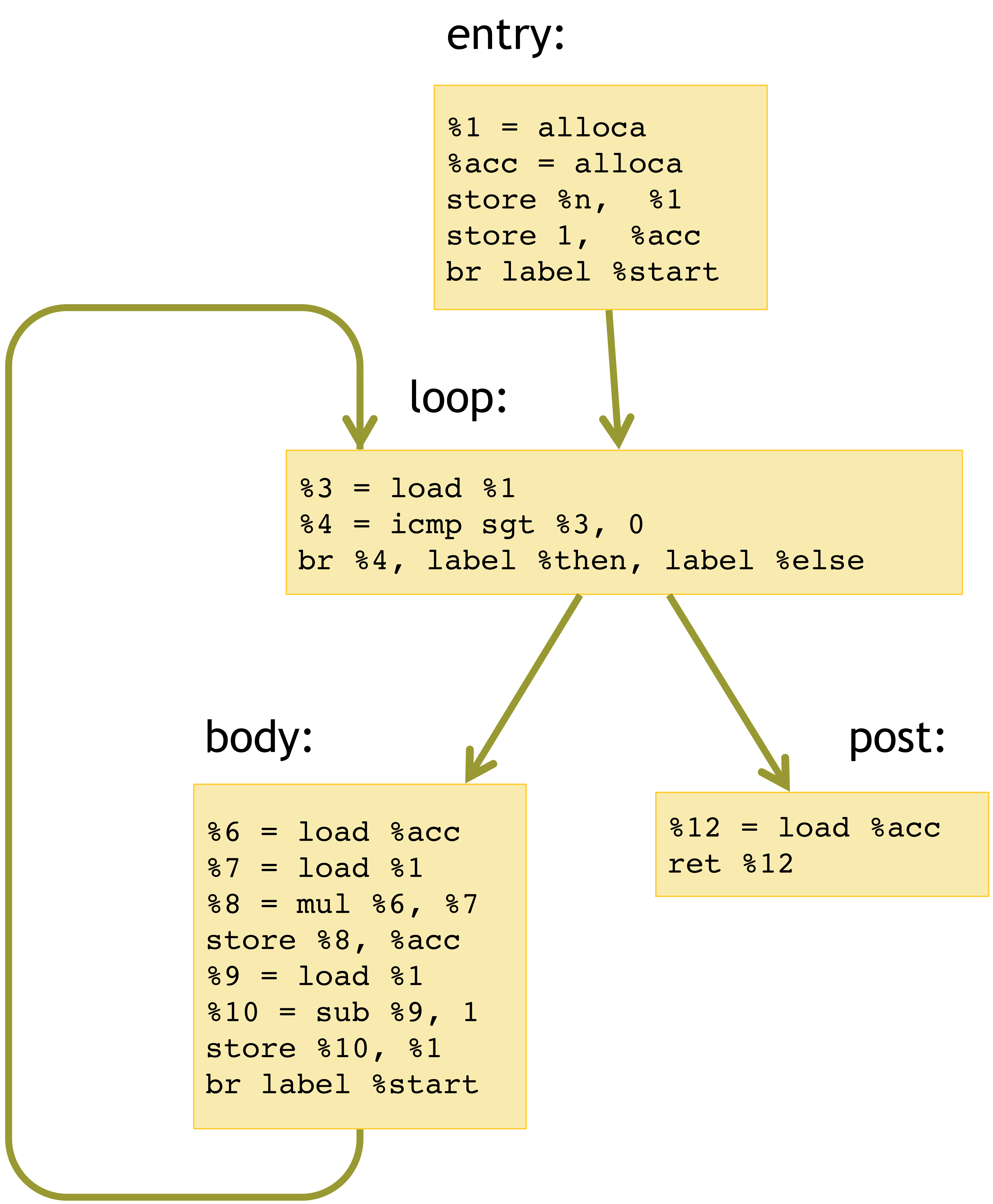
[Lattner et al.]



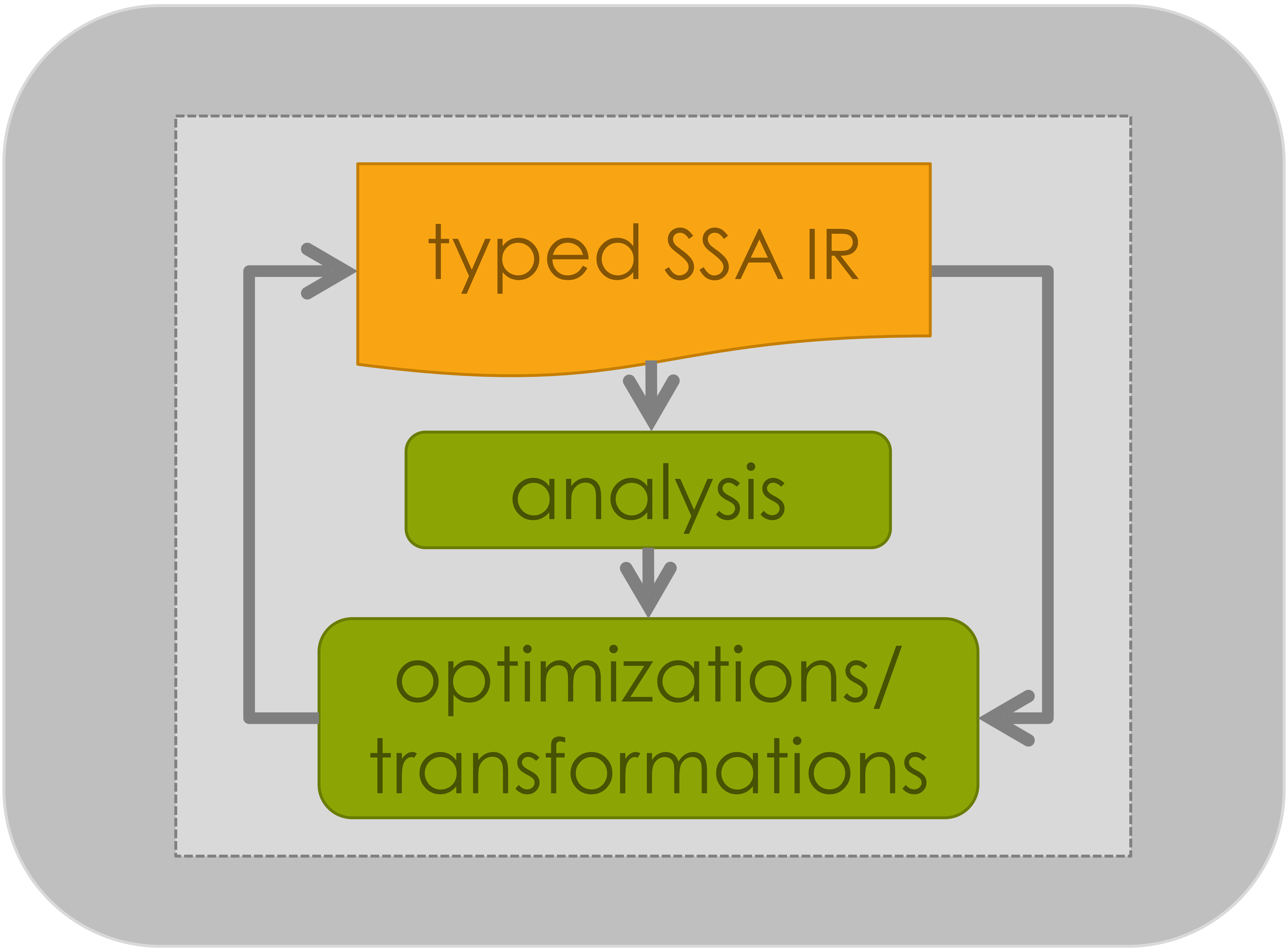
LLVM IR



LLVM IR



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



LLVM IR

Formal Semantics

Crellvm [Kang et al., 18]

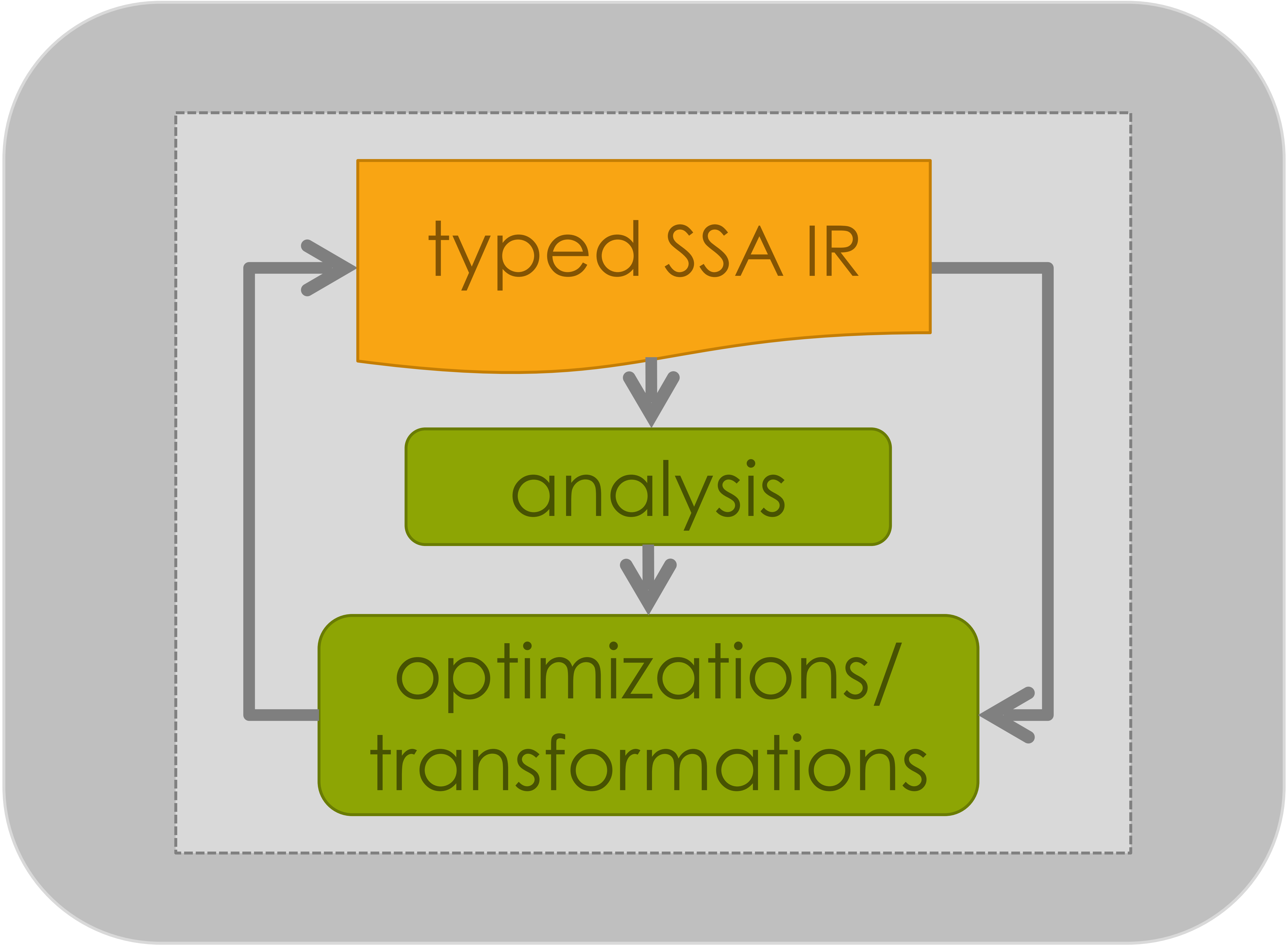
K-LLVM [Li and Gunter, 20]

Vellvm [Zhao et al., 12]

Taming UB [Lee et al., 17]

Concurrency [Chakraborty and Vafeiadis, 17]

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



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]

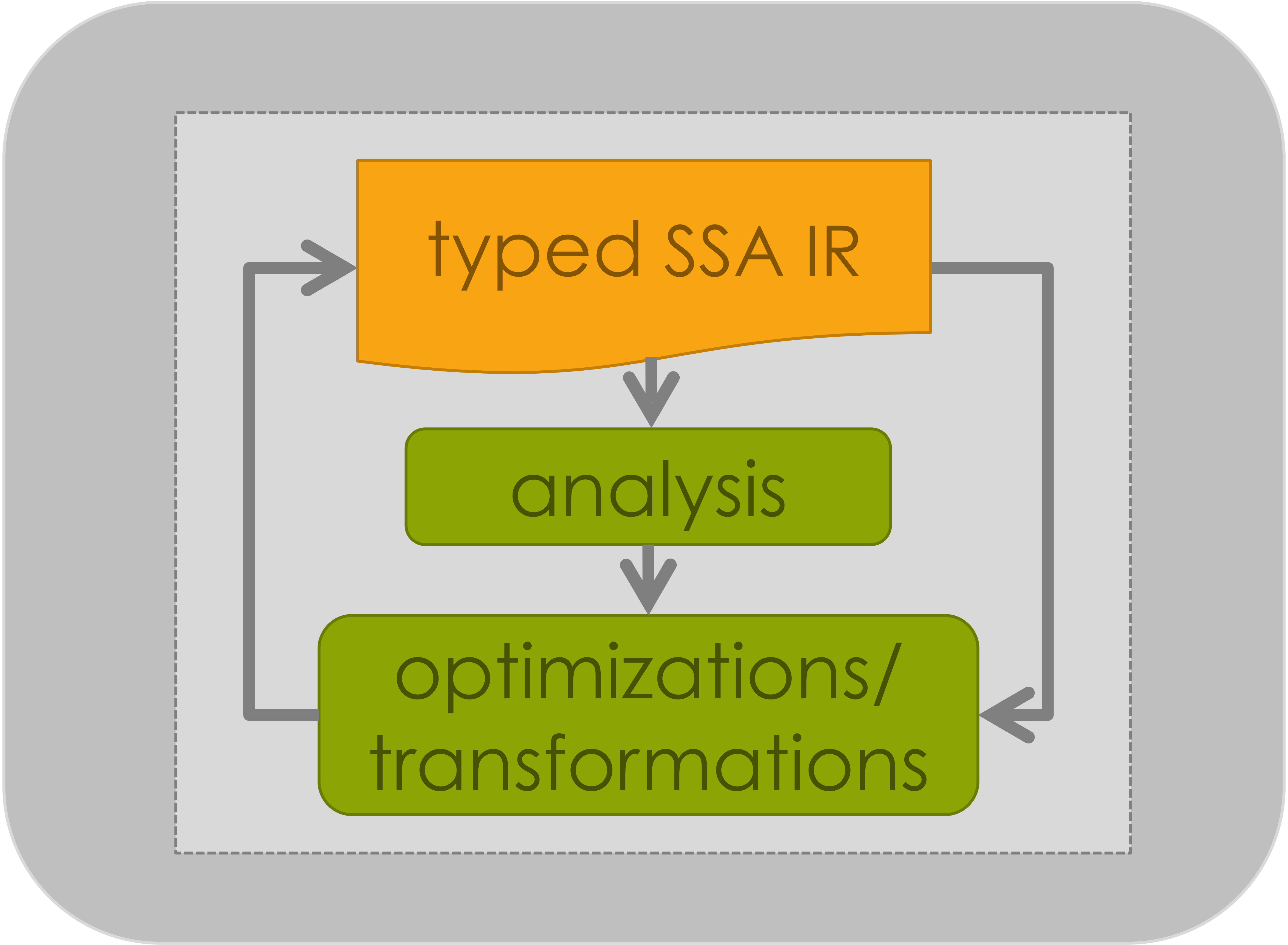
Concurrency [Chakraborty and Vafeiadis, 17]

Realistic Memory Models

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

Twin-Allocation [Lee et al., 18]

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



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]

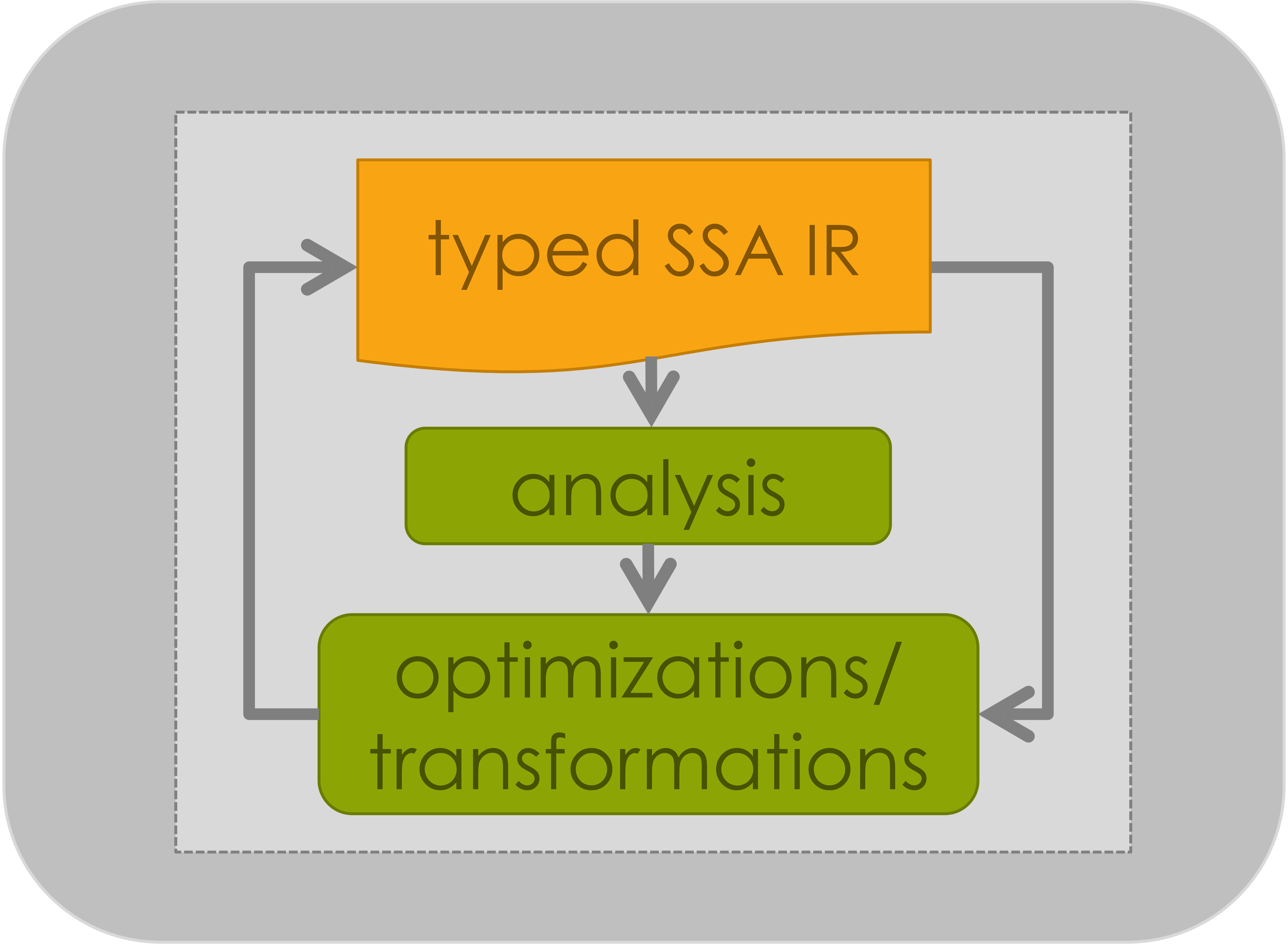
Twin-Allocation [Lee et al., 18]

Bug Finding

Alive [Lopes et al., 15]

Alive 2 [Lopes et al., 21]

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



LLVM IR

Formal Semantics

Crellvm [Kang et al., 18]

K-LLVM [Li and Gunter, 20]

Vellvm [Zhao et al., 12] ⟵ This work's ancestor

Taming UB [Lee et al., 17]

Concurrency [Chakraborty and Vafeiadis, 17]

Realistic Memory Models

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

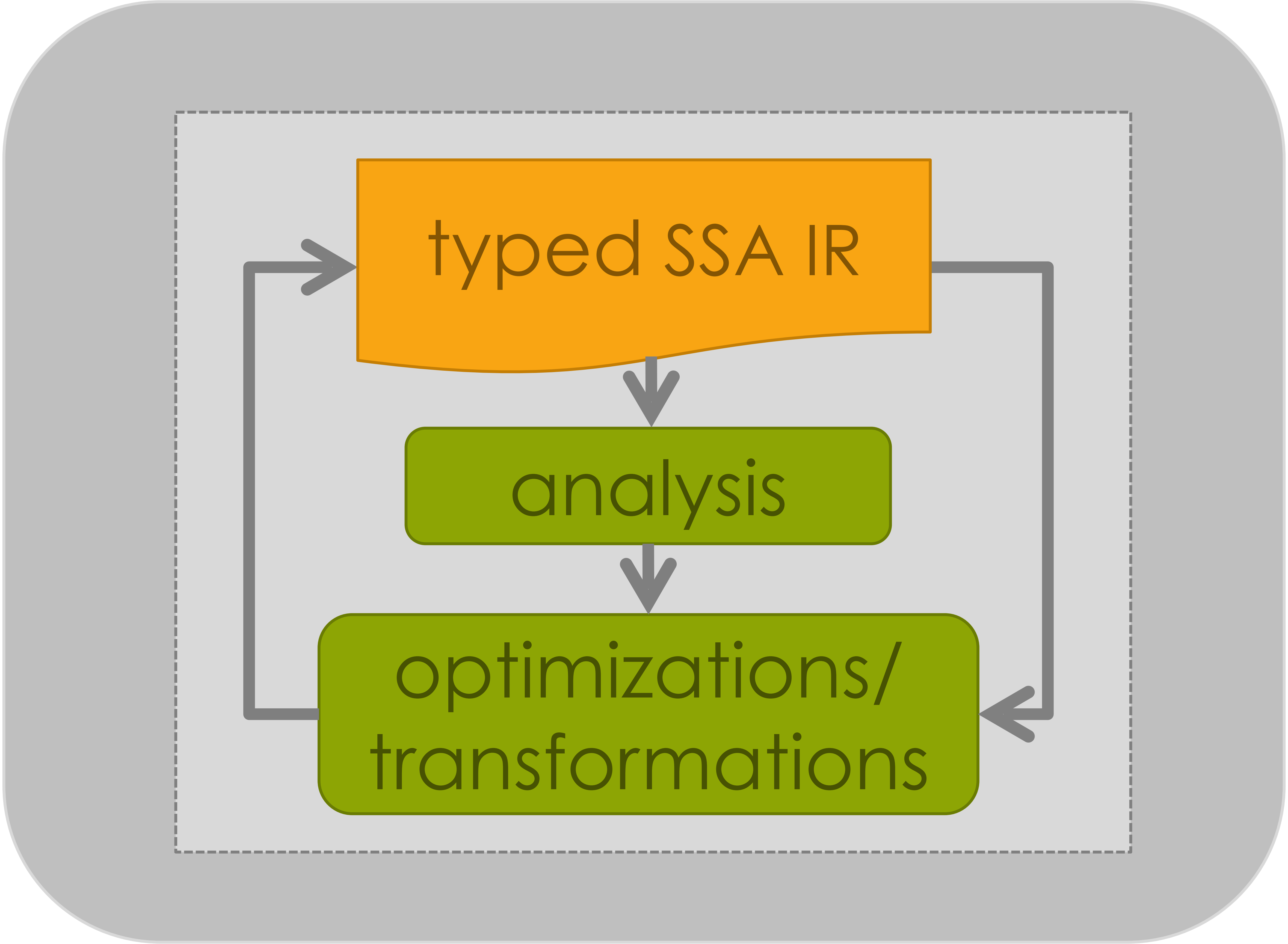
Twin-Allocation [Lee et al., 18]

Bug Finding

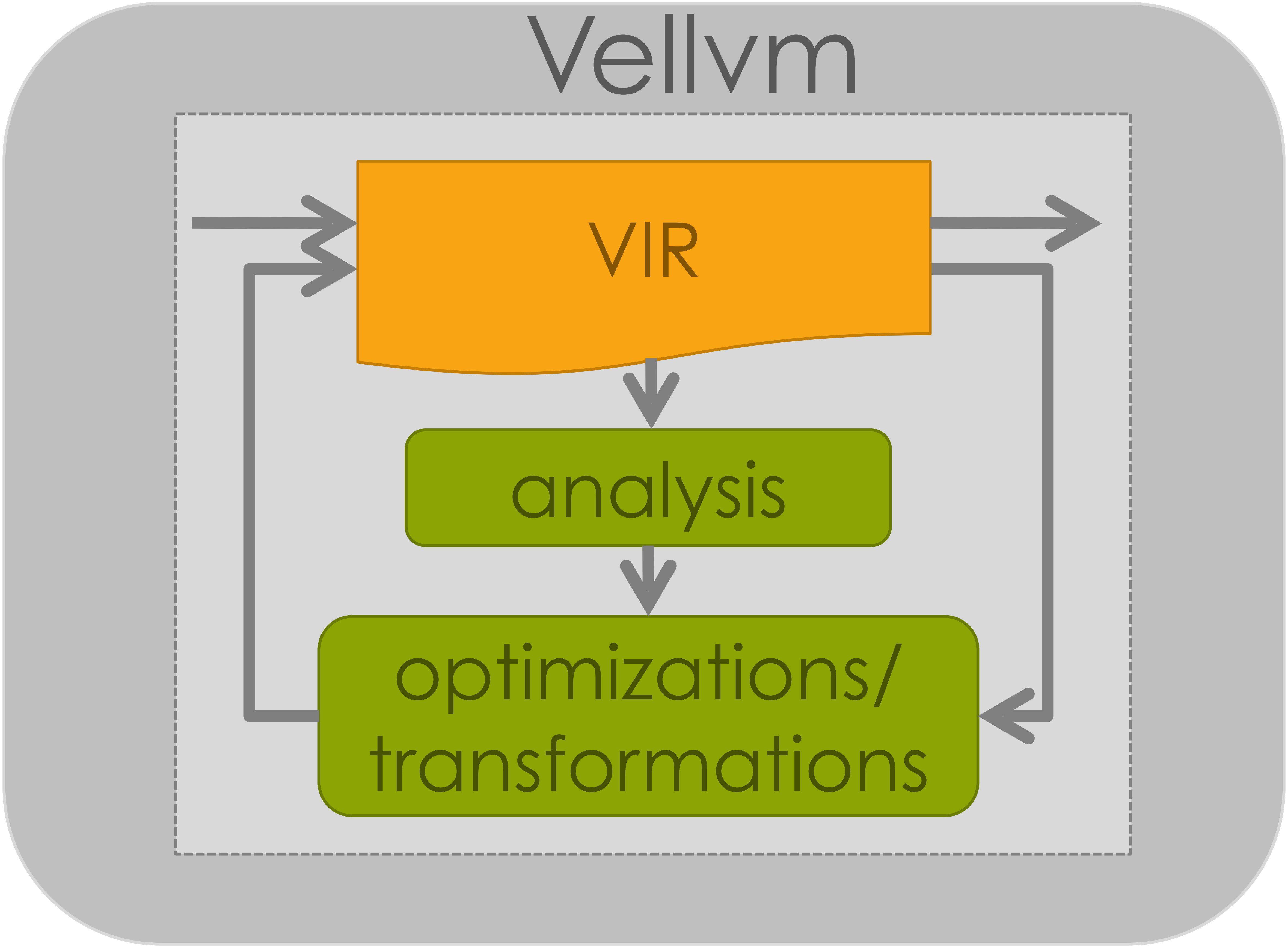
Alive [Lopes et al., 15]

Alive 2 [Lopes et al., 21]

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



The Vellvm Project



[Zhao and Zdancewic - CPP 2012]

Verified computation of dominators

[Zhao et al. - POPL 2012]

Formal semantics of IR + verified SoftBound

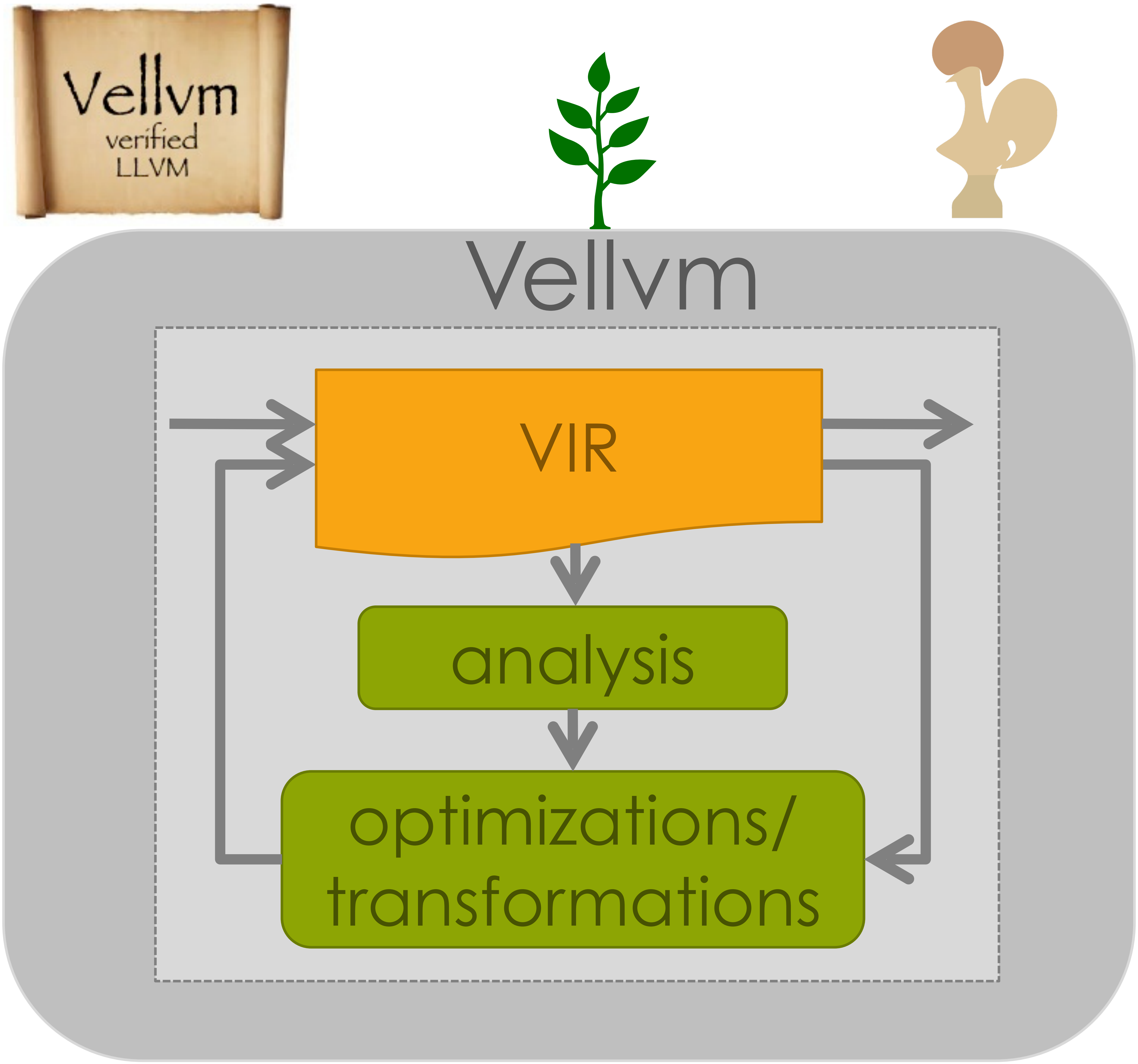
[Zhao et al. - PLDI 2013]

Verification of (v)mem2reg!

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

A success, but so monolithic it couldn't evolve!

The Vellvm Project



[Zhao and Zdancewic - CPP 2012]

Verified computation of dominators

[Zhao et al. - POPL 2012]

Formal semantics of IR + verified SoftBound

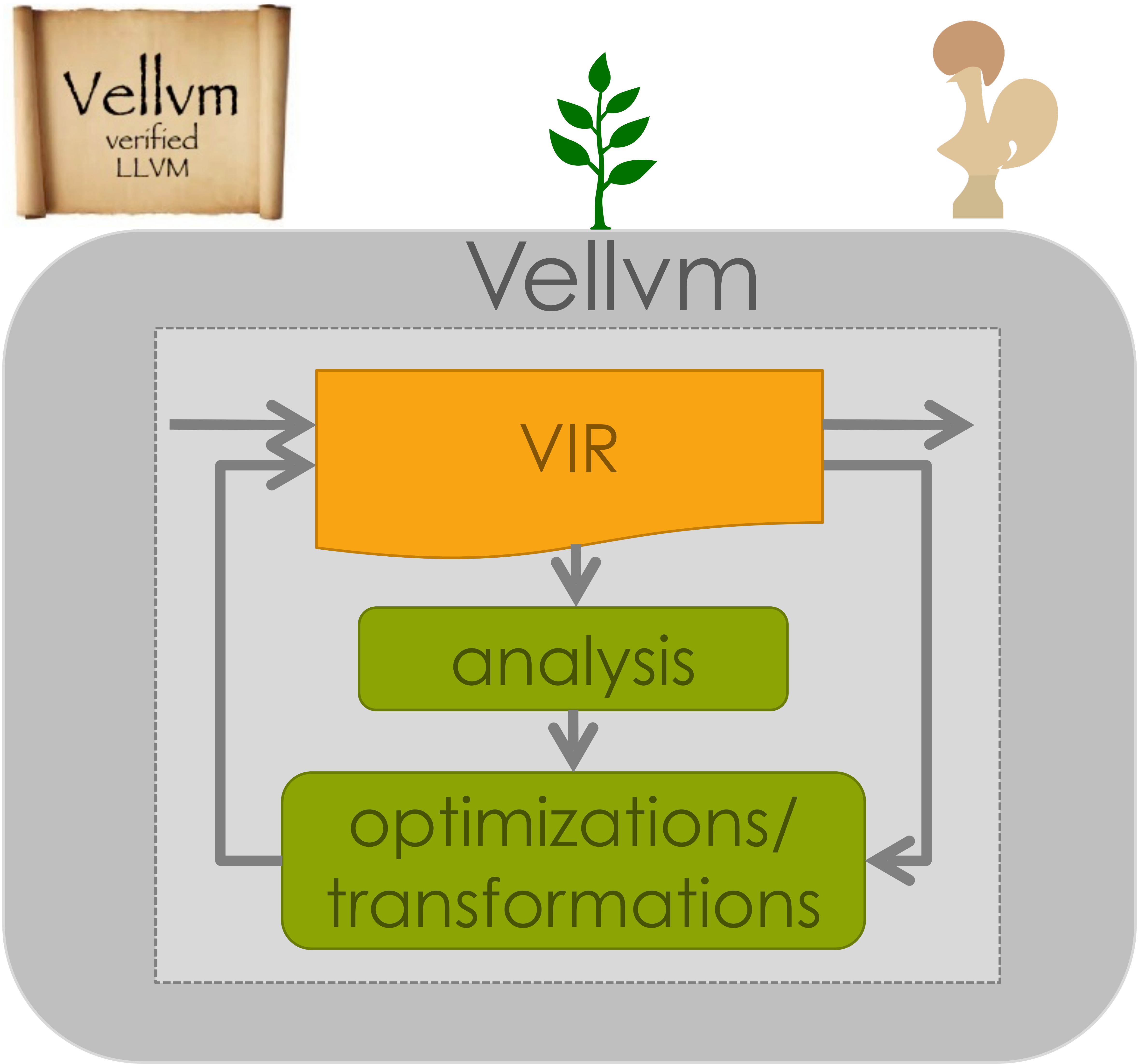
[Zhao et al. - PLDI 2013]

Verification of (v)mem2reg!

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

A success, but so monolithic it couldn't evolve!

The Vellvm Project



[Zhao and Zdancewic - CPP 2012]

Verified computation of dominators

[Zhao et al. - POPL 2012]

Formal semantics of IR + verified SoftBound

[Zhao et al. - PLDI 2013]

Verification of (v)mem2reg!

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

A success, but so monolithic it couldn't evolve!

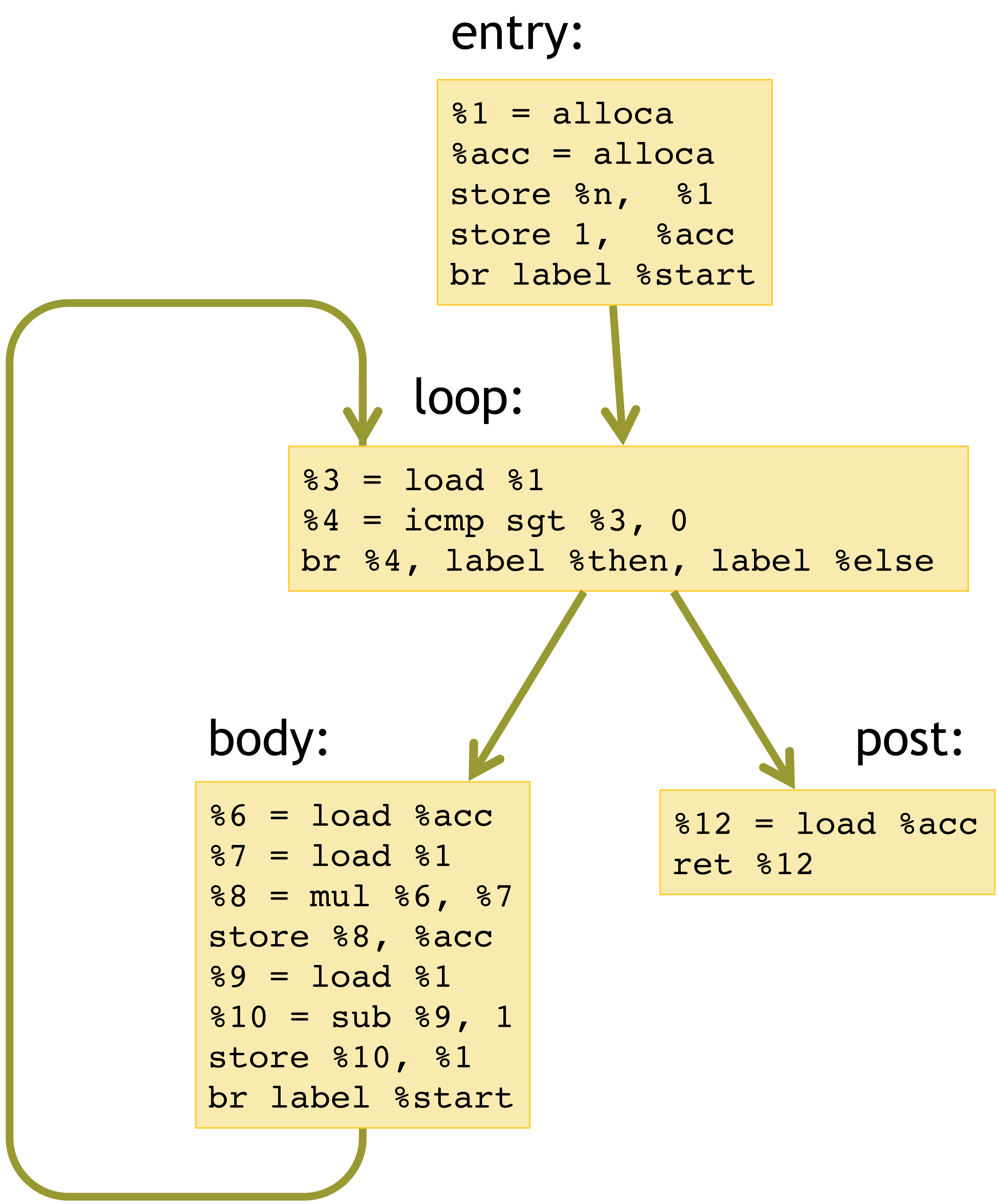
$$G \vdash pc, mem \rightarrow pc', mem'$$

Vellvm Legacy: Rough Approximation

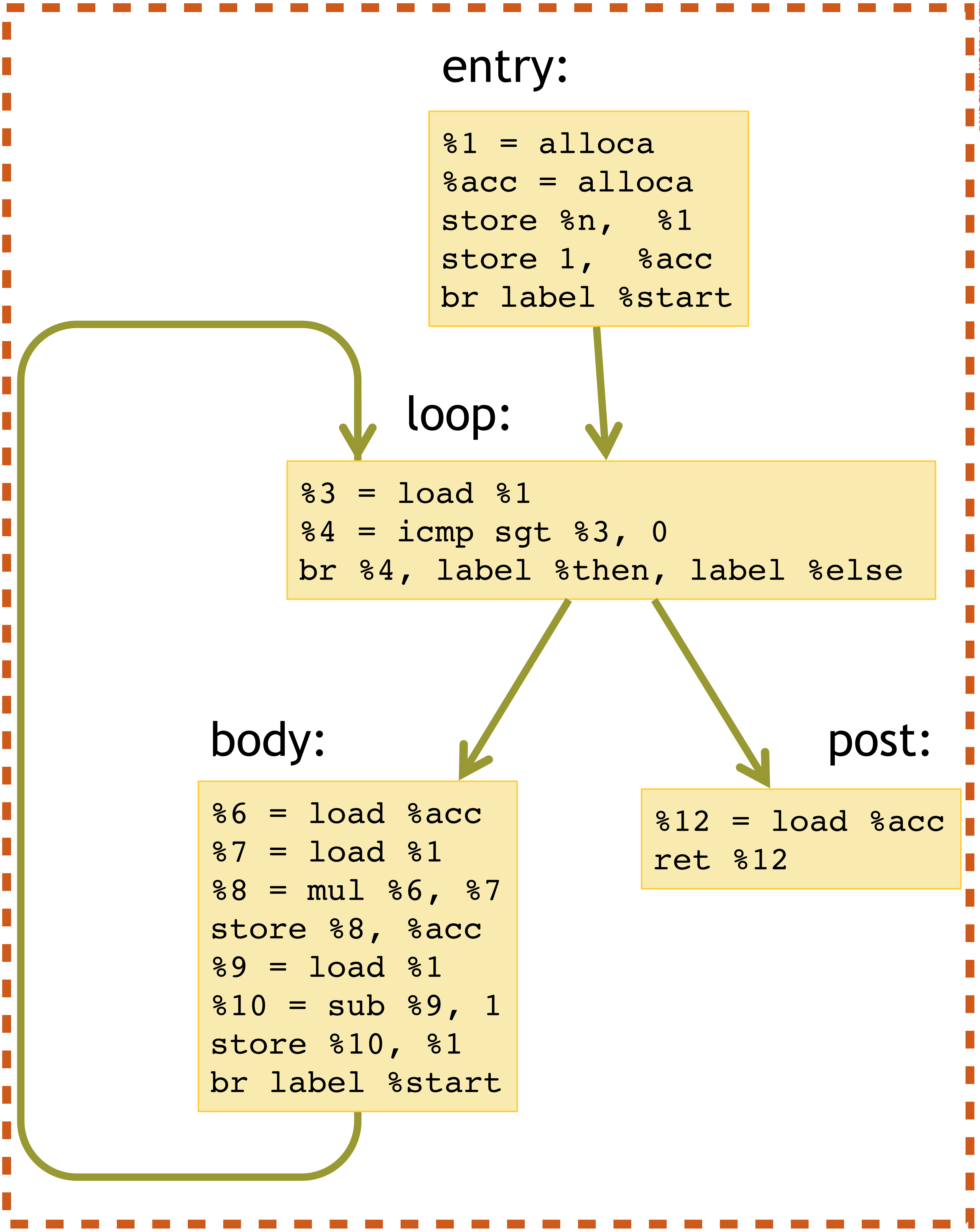
$$G \vdash pc,\ mem \rightarrow pc', mem'$$

Vellvm Legacy: Rough Approximation

$$G \vdash pc,\ mem \rightarrow pc', mem'$$

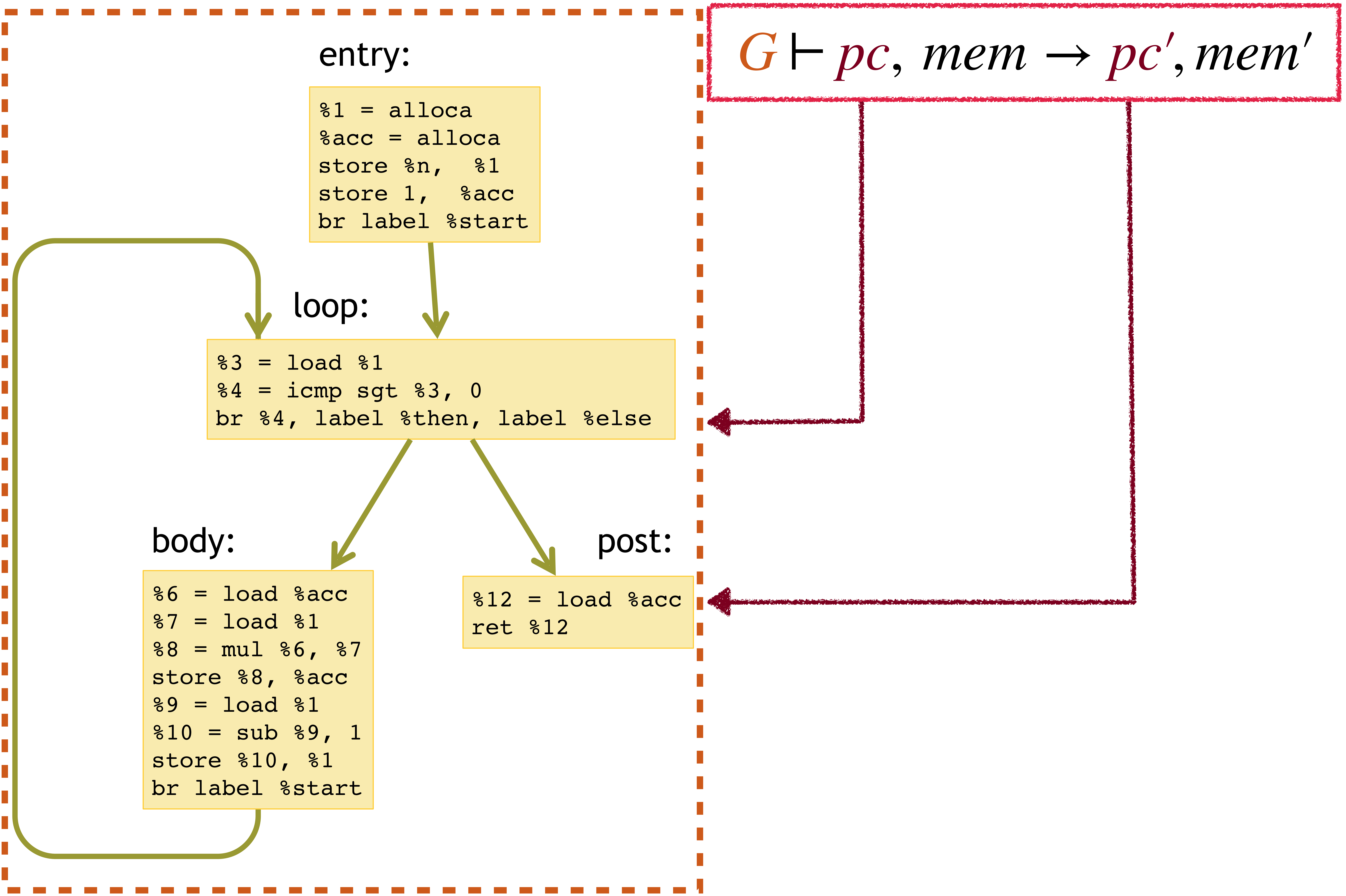


Vellvm Legacy: Rough Approximation

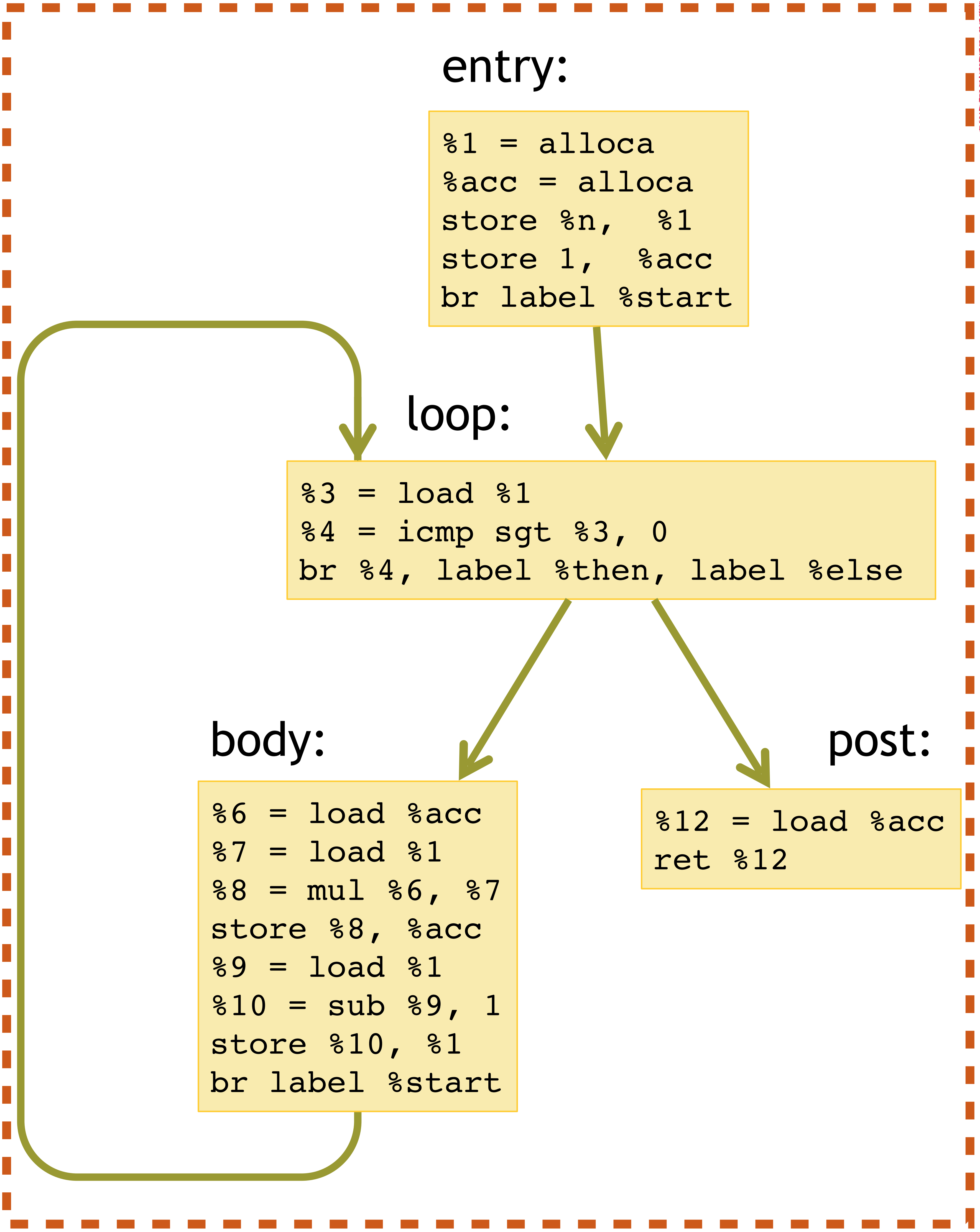


$$G \vdash pc,\ mem \rightarrow pc', mem'$$

Vellvm Legacy: Rough Approximation



Vellvm Legacy: Rough Approximation

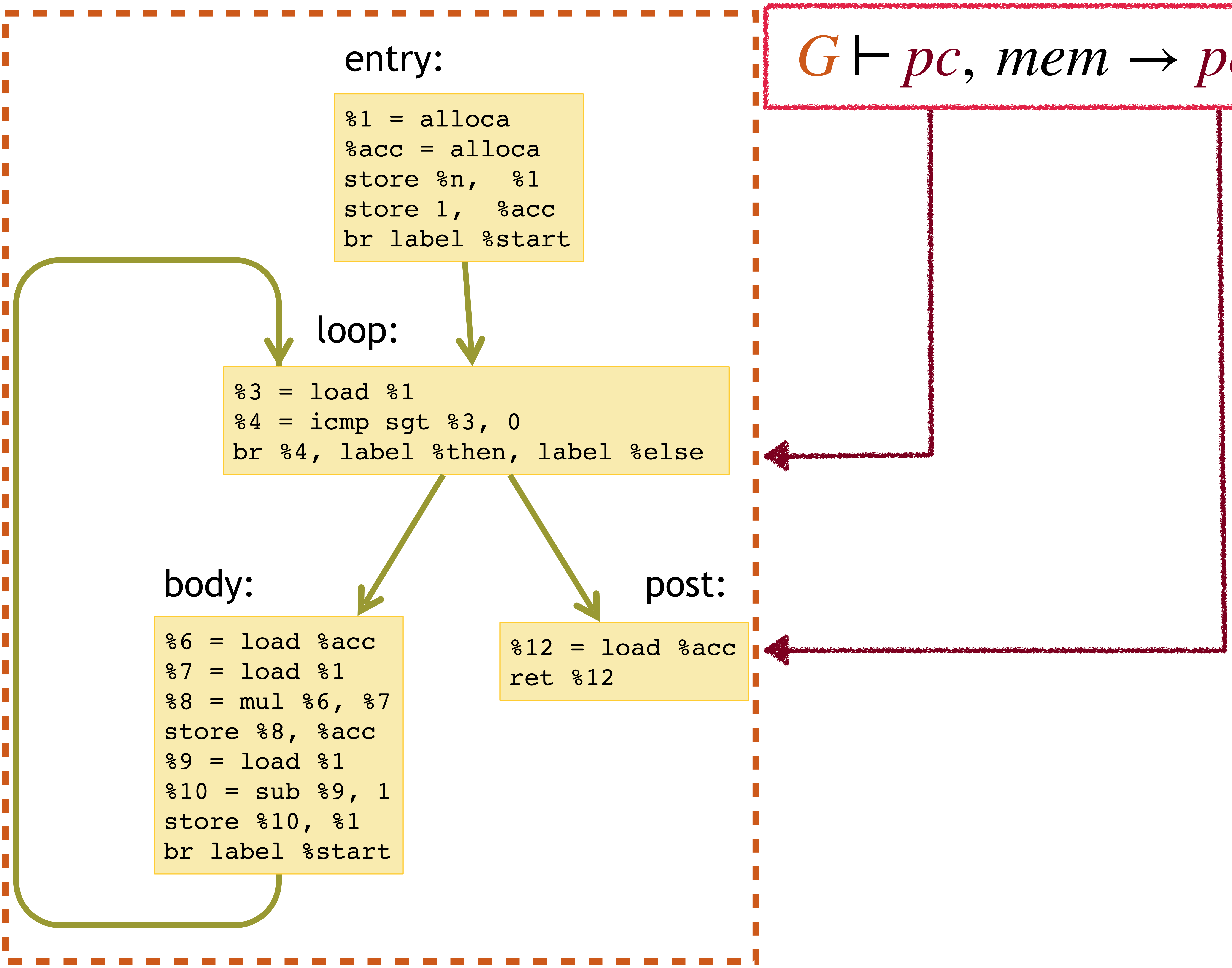


$$G \vdash pc,\, mem \rightarrow pc',\, mem'$$

Program transformations modify G, invariants must relate the pc

Lack of compositionality

Vellvm Legacy: Rough Approximation



$$G \vdash pc, mem \rightarrow pc', mem'$$

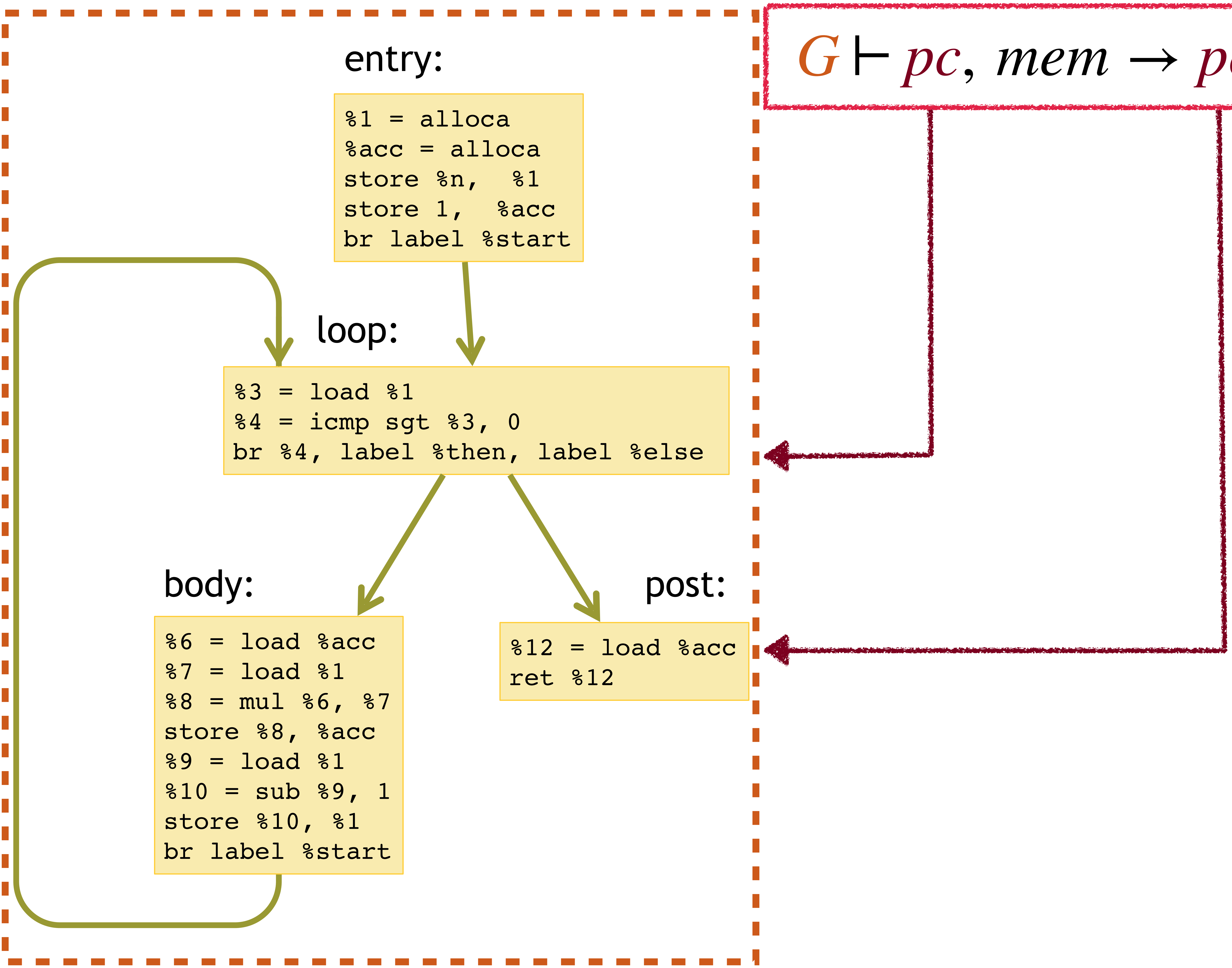
Program transformations modify G, invariants must relate the pc

Lack of compositionality

A single relation encompasses all aspects of the semantics

Lack of modularity

Vellvm Legacy: Rough Approximation



$G \vdash pc, mem \rightarrow pc', mem'$

Program transformations modify G, invariants must relate the pc

Lack of compositionality

A single relation encompasses all aspects of the semantics

Lack of modularity

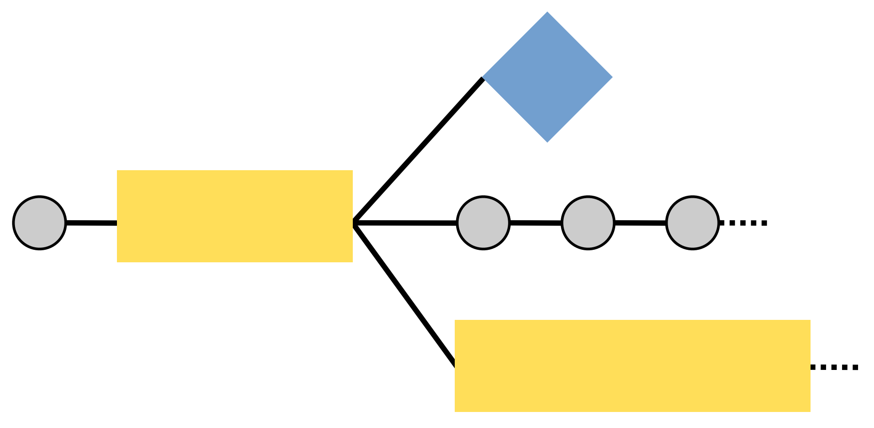
The semantics does not compute, it is a relation

Lack of executability

This Paper: a Redesign for Vellvm

[Xia et al. - POPL 2020]

Interaction Trees (itrees)



github.com/DeepSpec/InteractionTrees

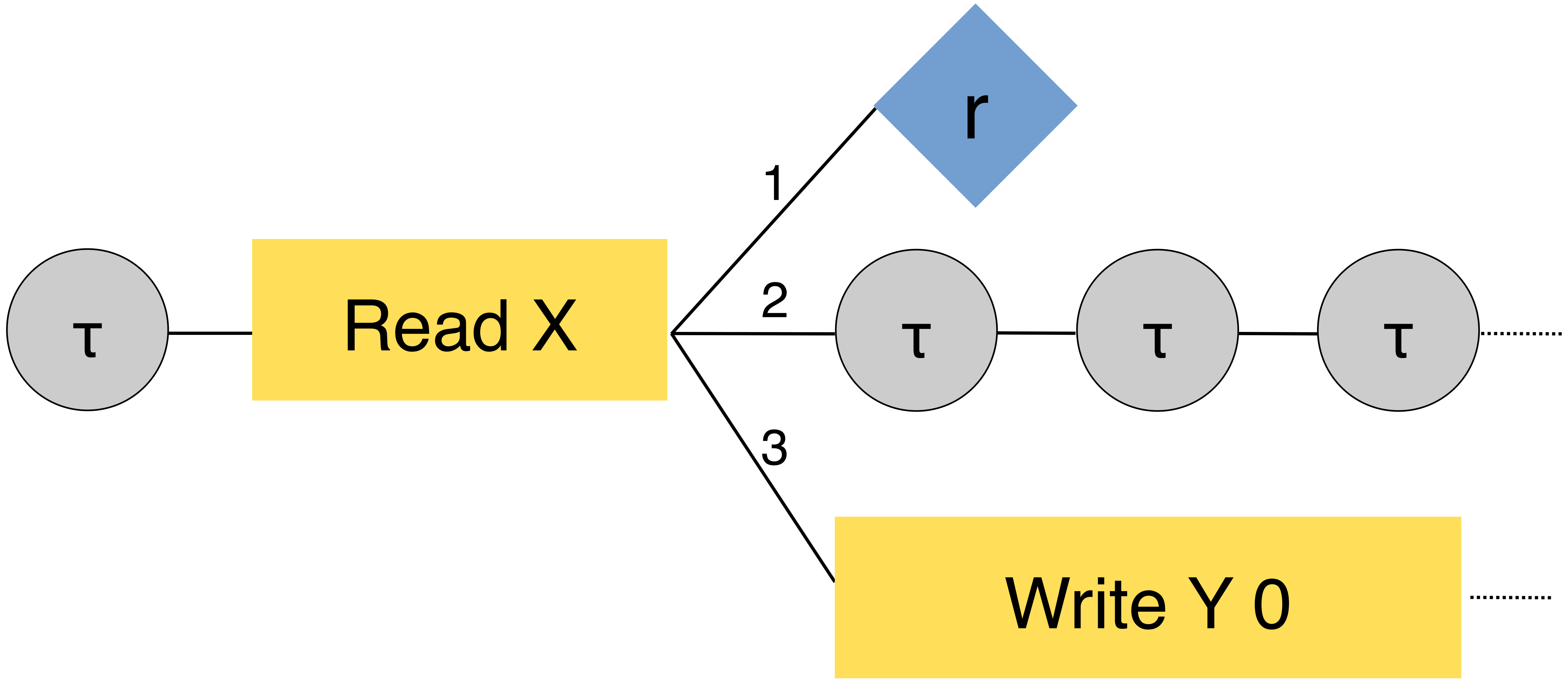
Used to build

[This paper]

(Re)Vellvm

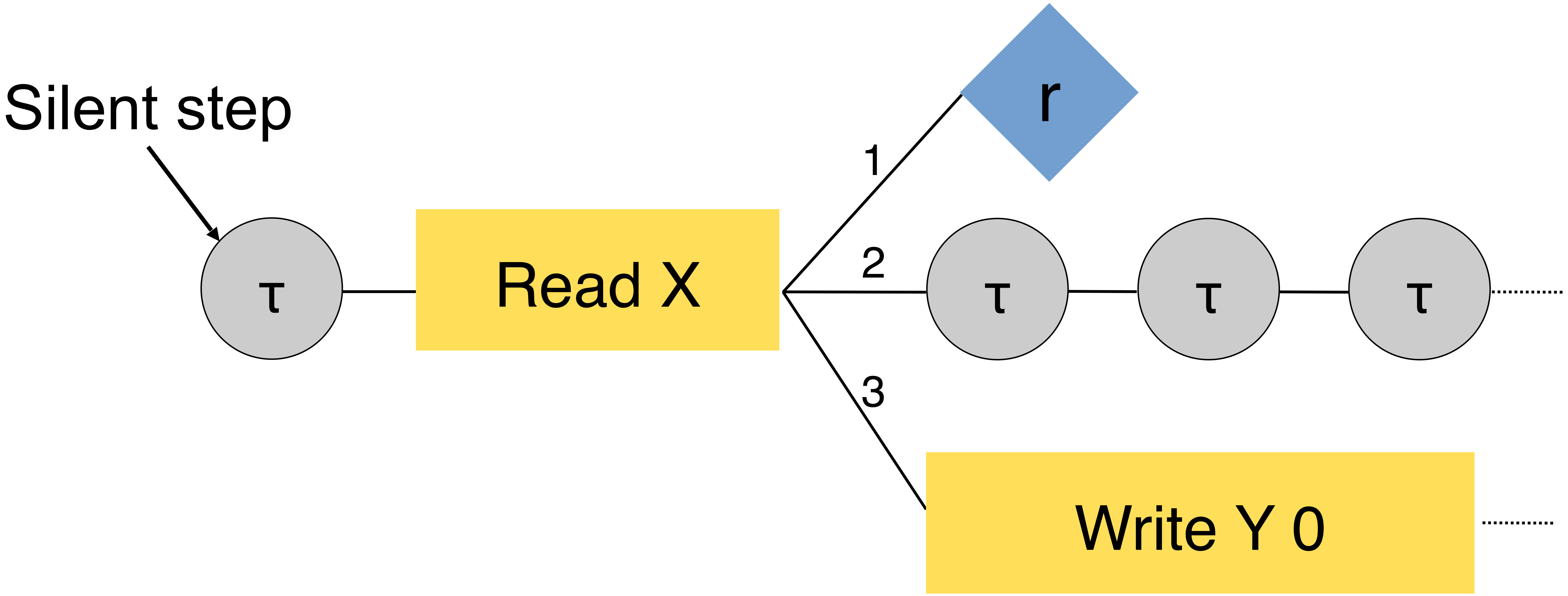
github.com/vellvm/vellvm

A Tree Represents an Interactive Computation



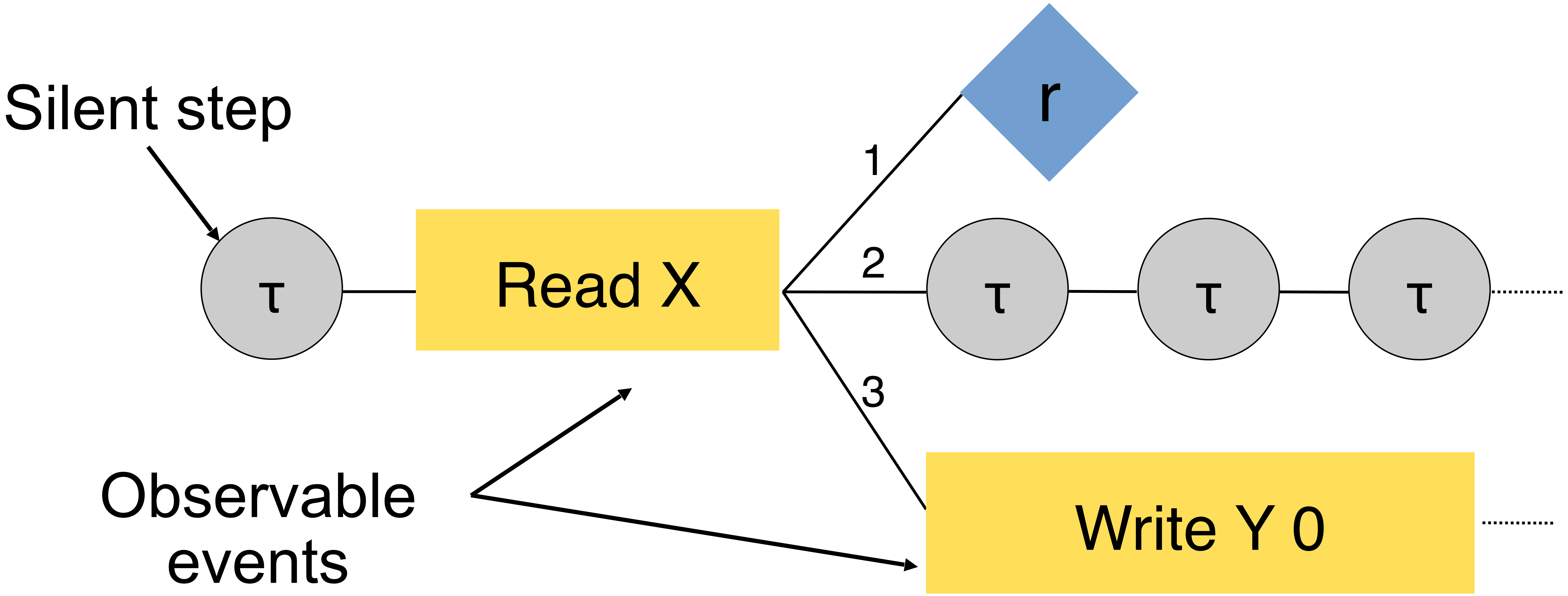
We consider here a computation whose interactions with the environment are read and writes to a state

A Tree Represents an Interactive Computation



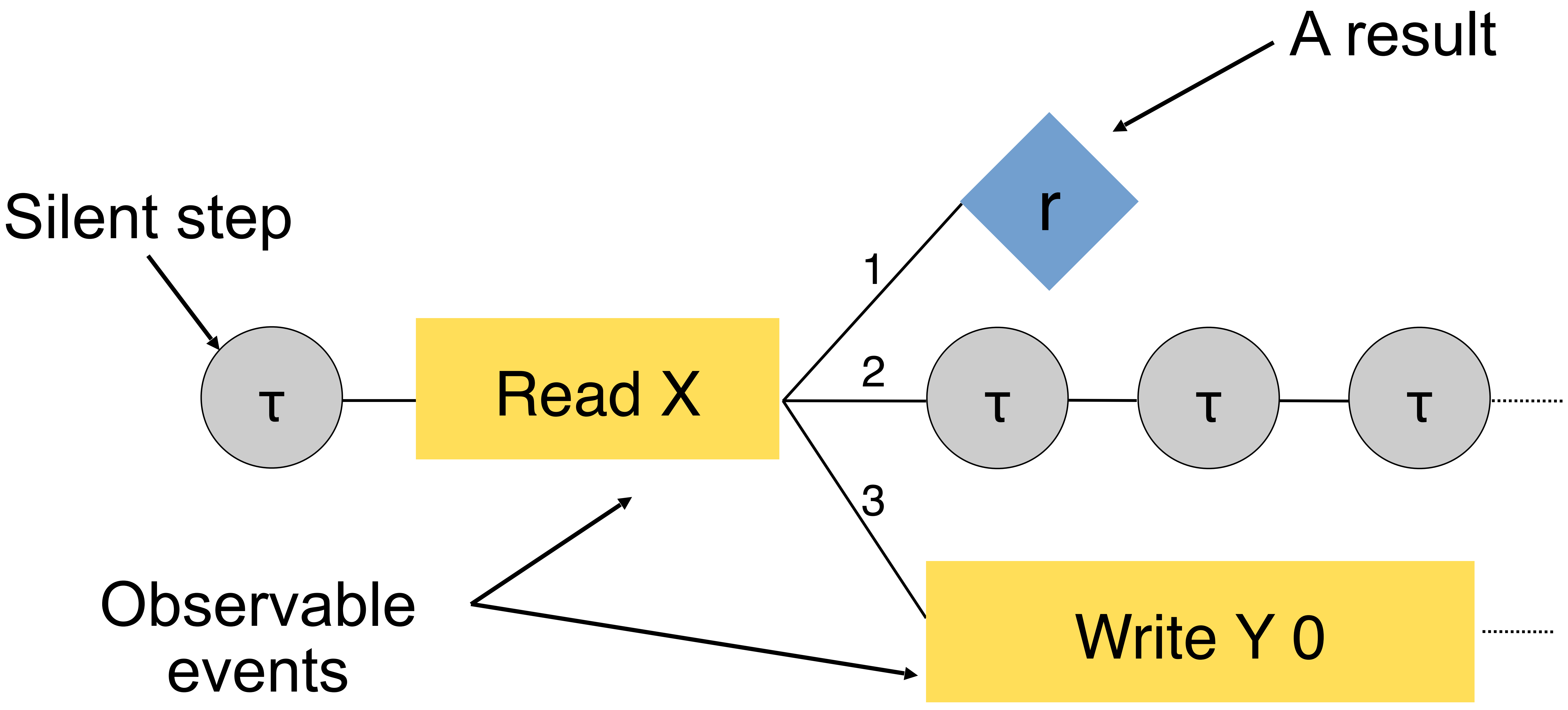
We consider here a computation whose interactions with the environment are read and writes to a state

A Tree Represents an Interactive Computation



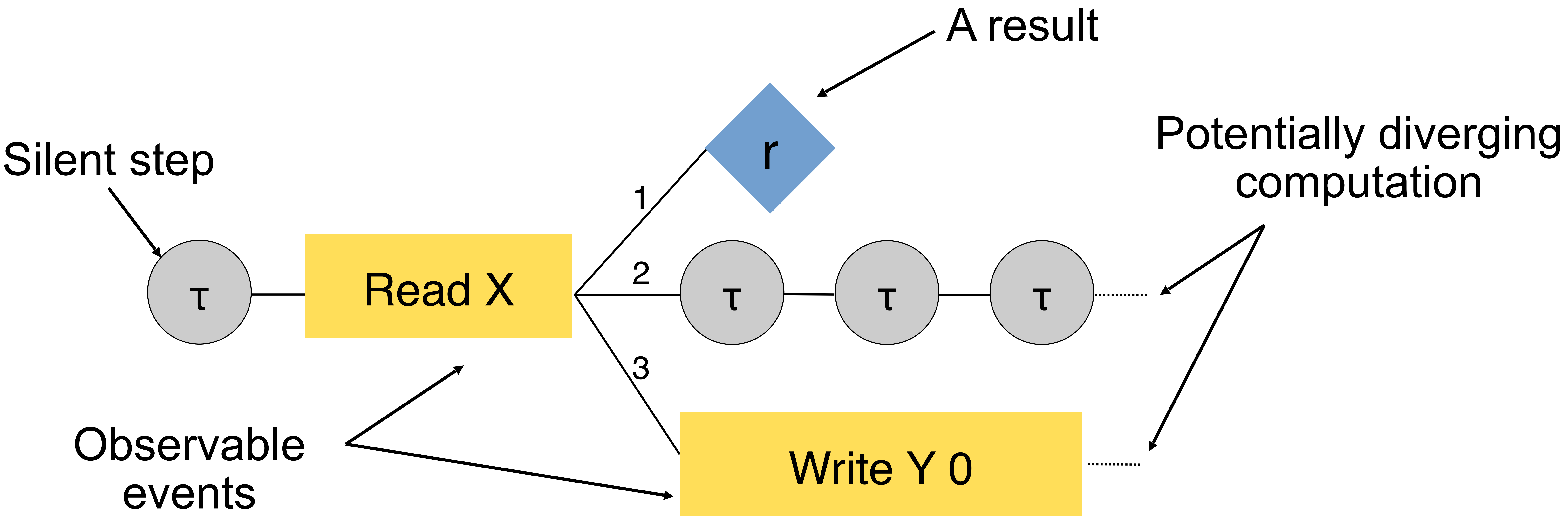
We consider here a computation whose interactions with the environment are read and writes to a state

A Tree Represents an Interactive Computation



We consider here a computation whose interactions with the environment are read and writes to a state

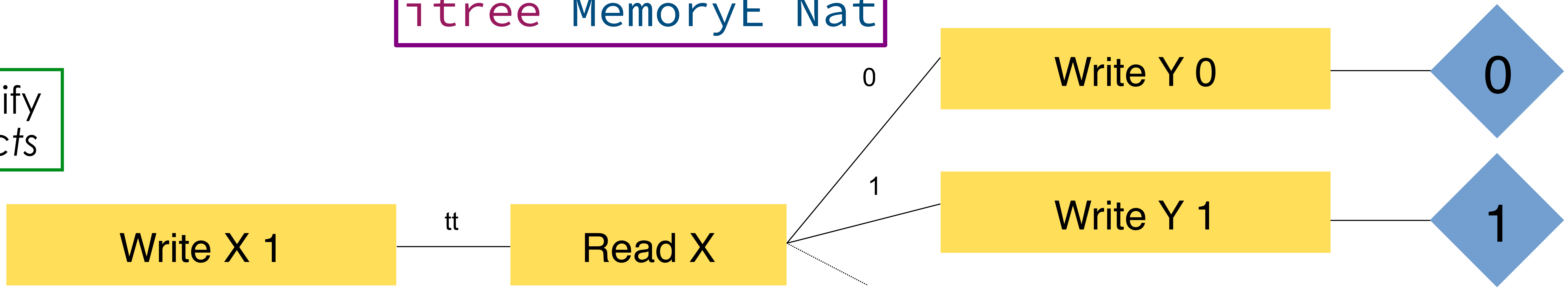
A Tree Represents an Interactive Computation



We consider here a computation whose interactions with the environment are read and writes to a state

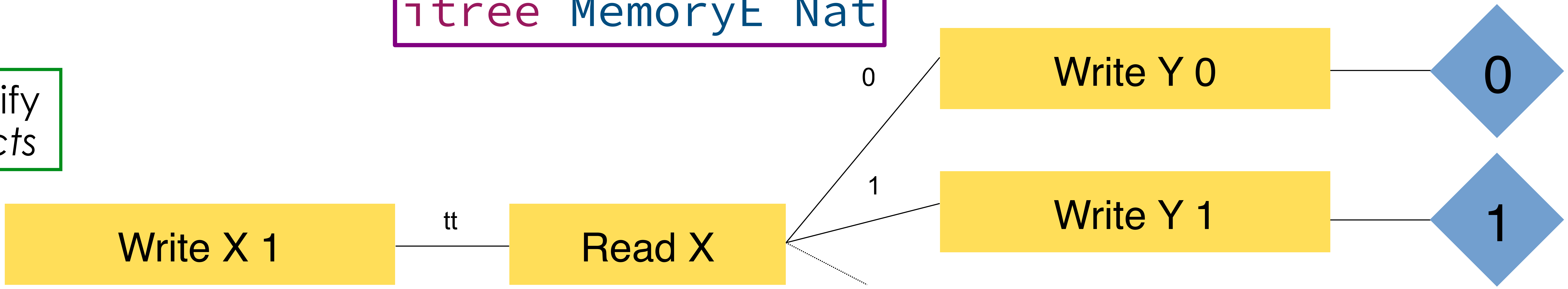
Events only specify the type of *effects*

`itree MemoryE Nat`



`itree MemoryE Nat`

Events only specify the type of *effects*



generic, provided by the library

interp handle

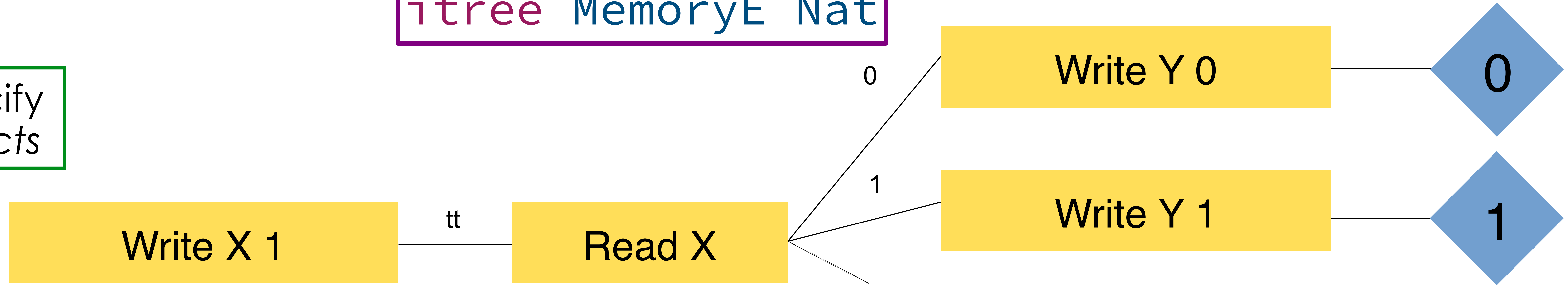
The tree is interpreted via an event handler

specific, user defined

The semantics of effects is introduced

```
itree MemoryE Nat
```

Events only specify the type of *effects*



generic, provided by the library

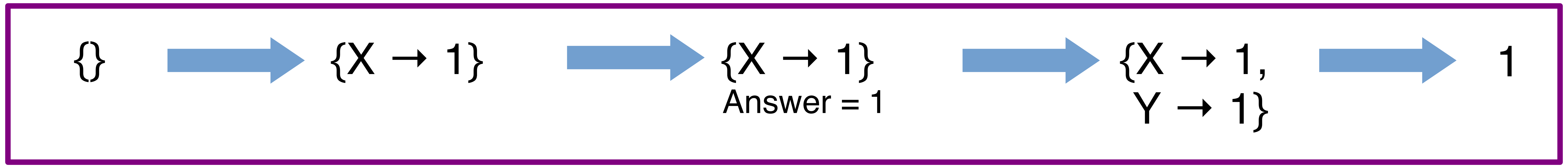
interp handle

The tree is interpreted via an event handler

specific, user defined

The semantics of effects is introduced

```
St -> itree voidE (St * Nat)
```

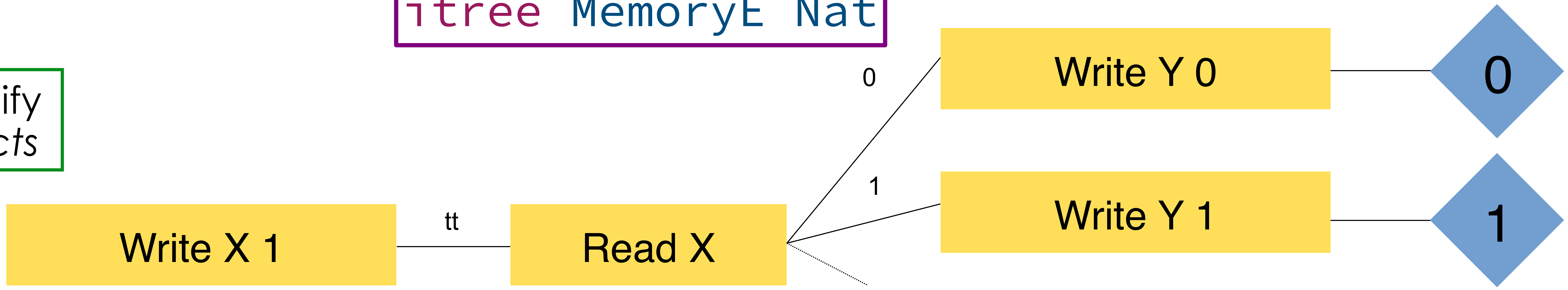


```
X = 1;
Y = X;
ret X
```

represented as

`itree MemoryE Nat`

Events only specify the type of *effects*



generic, provided by the library

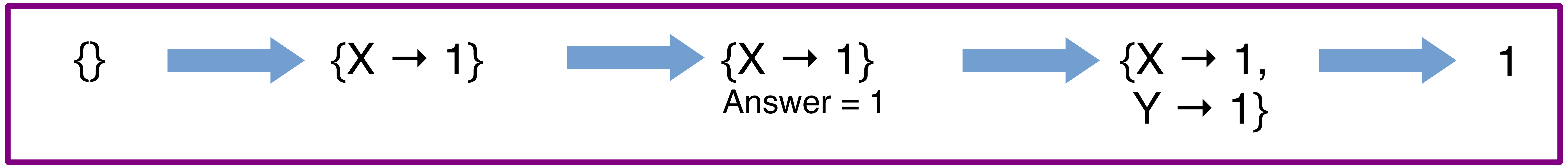
interp handle

The tree is interpreted via an event handler

specific, user defined

The semantics of effects is introduced

`St -> itree voidE (St * Nat)`



A general recipe

A general recipe

1. Write down the *syntax* $\mathscr{L}$ of your language

A general recipe

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$

A general recipe

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$

A general recipe

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$

Compositionality

A general recipe

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$

   Compositionality
4. *Handle* $\mathscr{E}$ into an appropriate monad $\mathscr{M}$, get an *interpreter* for whole programs for free

A general recipe

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$ — Compositionality
4. *Handle* $\mathscr{E}$ into an appropriate monad $\mathscr{M}$, get an *interpreter* for whole programs for free — Modularity

A general recipe

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$
4. *Handle* $\mathscr{E}$ into an appropriate monad $\mathscr{M}$, get an *interpreter* for whole programs for free
5. As a bonus, extract the result to OCaml to get a *definitional interpreter*

Compositionality

Modularity

A general recipe

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$ — Compositionality
4. *Handle* $\mathscr{E}$ into an appropriate monad $\mathscr{M}$, get an *interpreter* for whole programs for free — Modularity
5. As a bonus, extract the result to OCaml to get a *definitional interpreter* — Executability

Scaling to a Fully Fledged Language

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$
4. *Handle* $\mathscr{E}$ into an appropriate monad $\mathscr{M}$, get an *interpreter* for whole programs for free
5. As a bonus, extract the result to OCaml to get a *definitional interpreter*

Event interface for an IR program

$$E = L_E +' G_E +' M_E$$
$$+' Call_E +' Intrinsics_E$$
$$+' Pick_E +' UB_E$$
$$+' Debug_E +' Failure_E$$

Scaling to a Fully Fledged Language

1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$
4. *Handle* $\mathscr{E}$ into an appropriate monad $\mathscr{M}$, get an *interpreter* for whole programs for free
5. As a bonus, extract the result to OCaml to get a *definitional interpreter*

Event interface for an IR program

$$E = L_E +' G_E +' M_E$$
$$+' Call_E +' Intrinsics_E$$
$$+' Pick_E +' UB_E$$
$$+' Debug_E +' Failure_E$$

Compositional representation for (open) IR programs

Scaling to a Fully Fledged Language

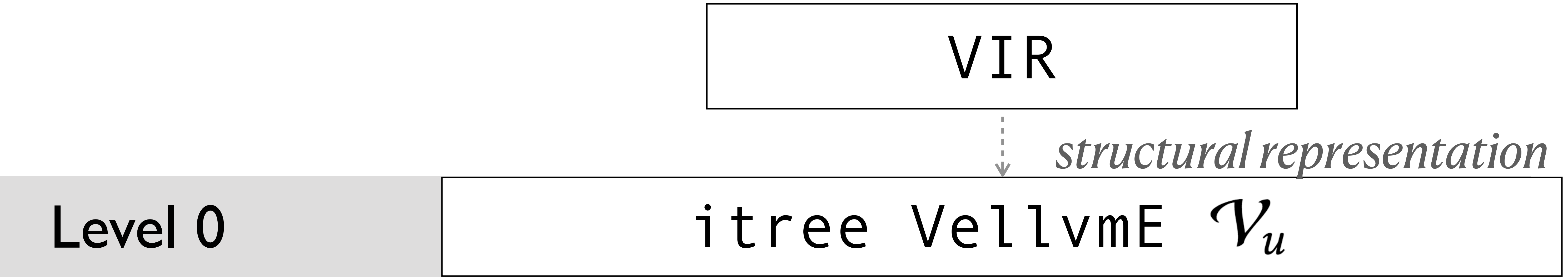
1. Write down the *syntax* $\mathscr{L}$ of your language
2. Inventory the *effects* of your language and write the corresponding *event interface* $\mathscr{E}$
3. Use the itree combinators to *represent* $\mathscr{L}$ as *non-interpreted itrees* over $\mathscr{E}$
4. *Handle* $\mathscr{E}$ into an appropriate monad $\mathscr{M}$, get an *interpreter* for whole programs for free
5. As a bonus, extract the result to OCaml to get a *definitional interpreter*

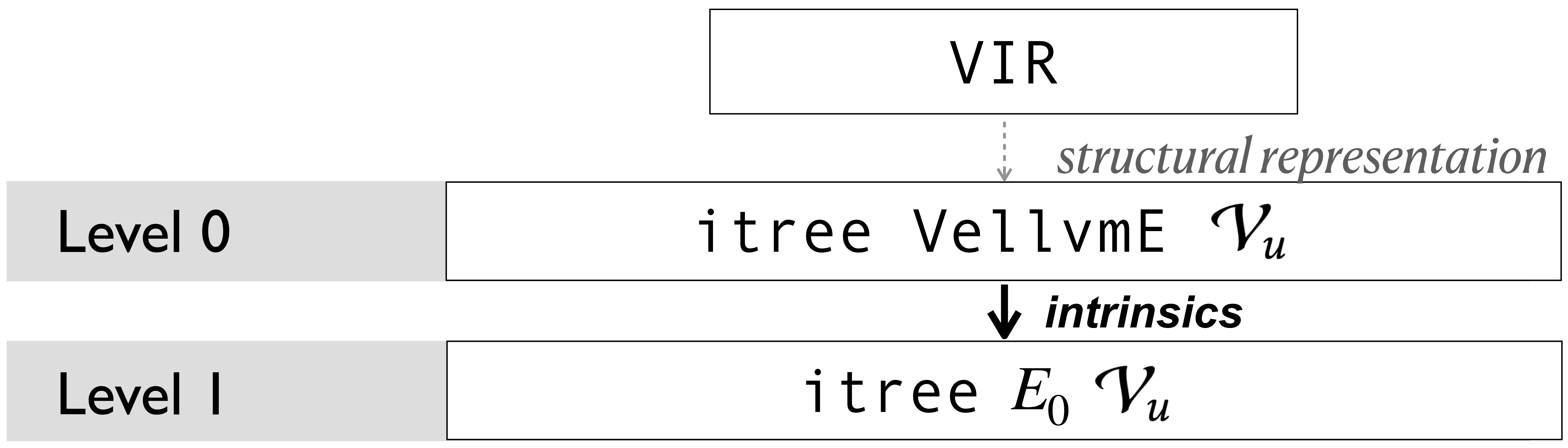
Event interface for an IR program

$$E = L_E +' G_E +' M_E +' Call_E +' Intrinsics_E +' Pick_E +' UB_E +' Debug_E +' Failure_E$$

Compositional representation for (open) IR programs

VIR


VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$


VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
Level 1
itree E_0 $\mathcal{V}_u$

VIR

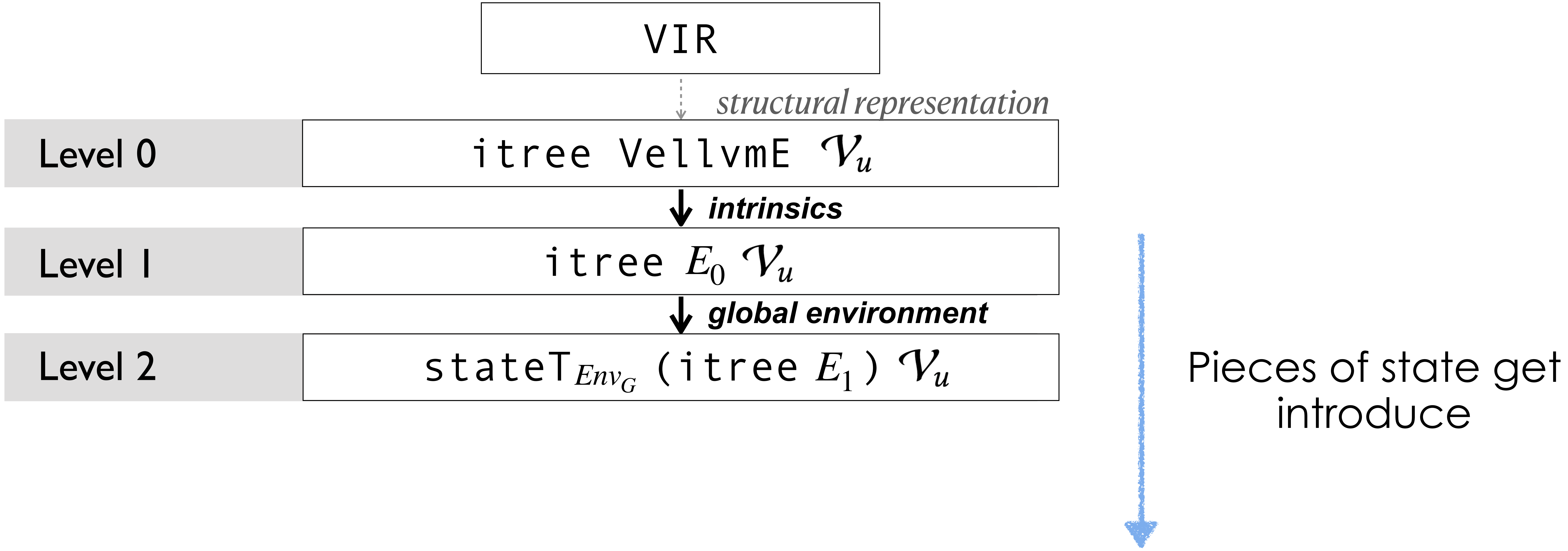
structural representation

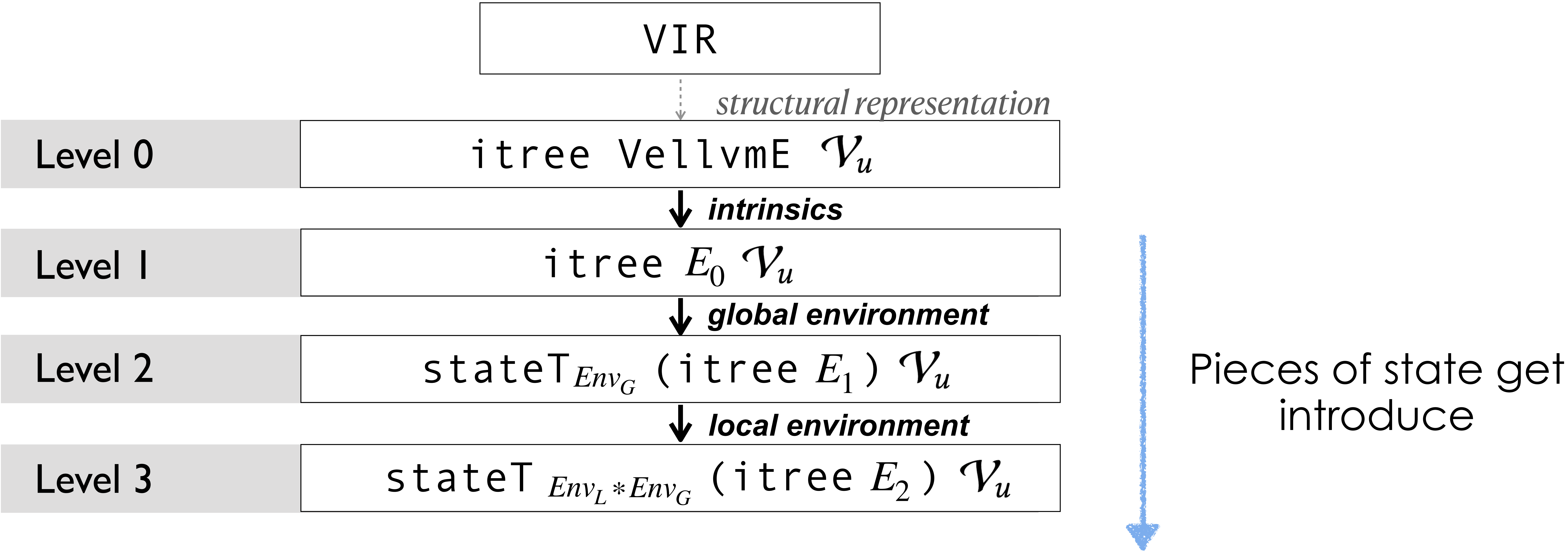
Level 0 | `itree VellvmE` $\mathcal{V}_u$

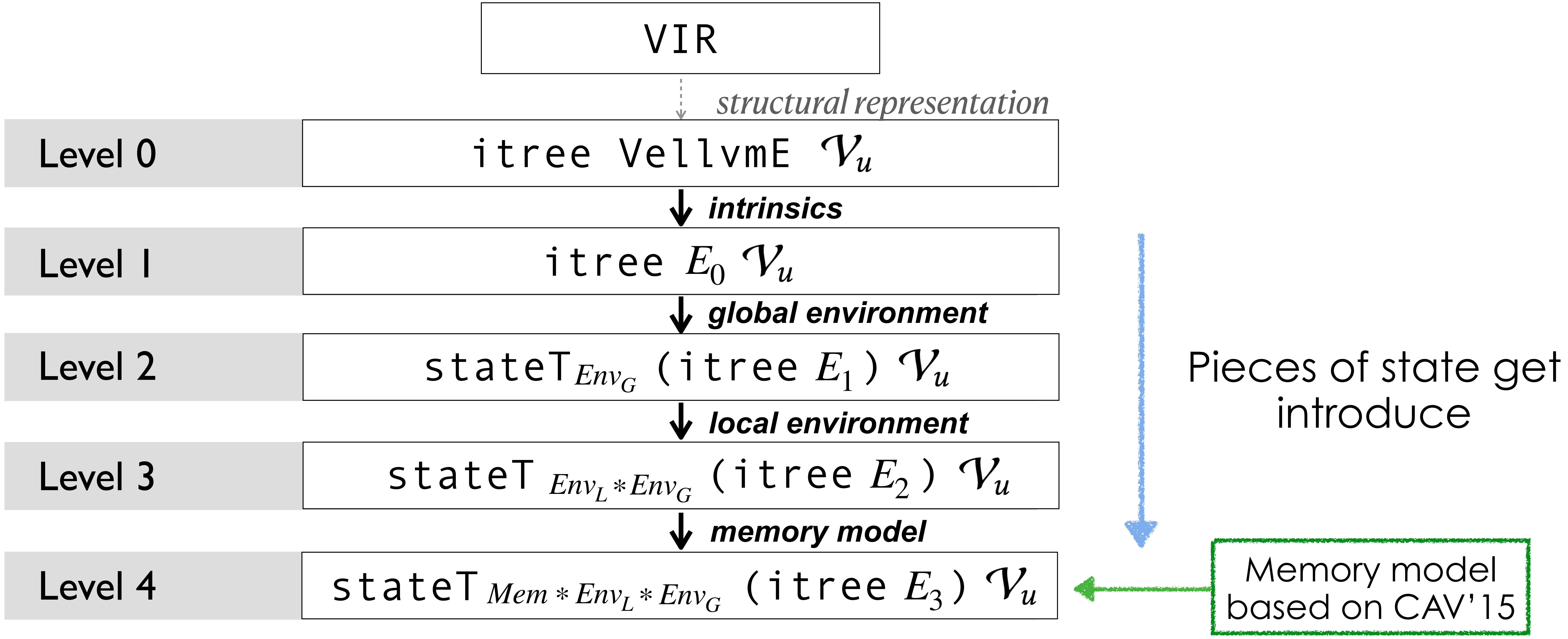
intrinsics

Level 1 | `itree` E_0 $\mathcal{V}_u$

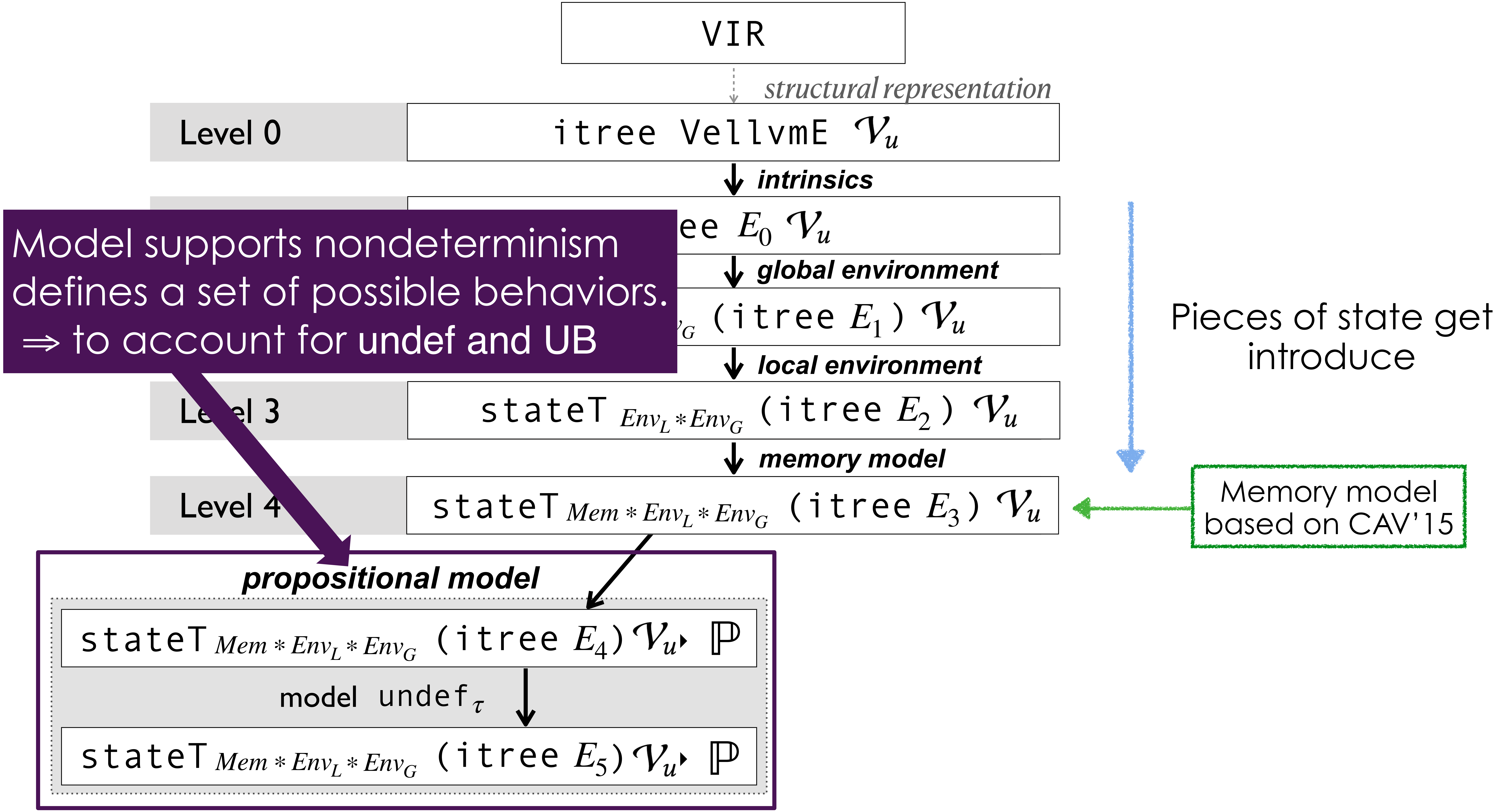
Pieces of state get introduce

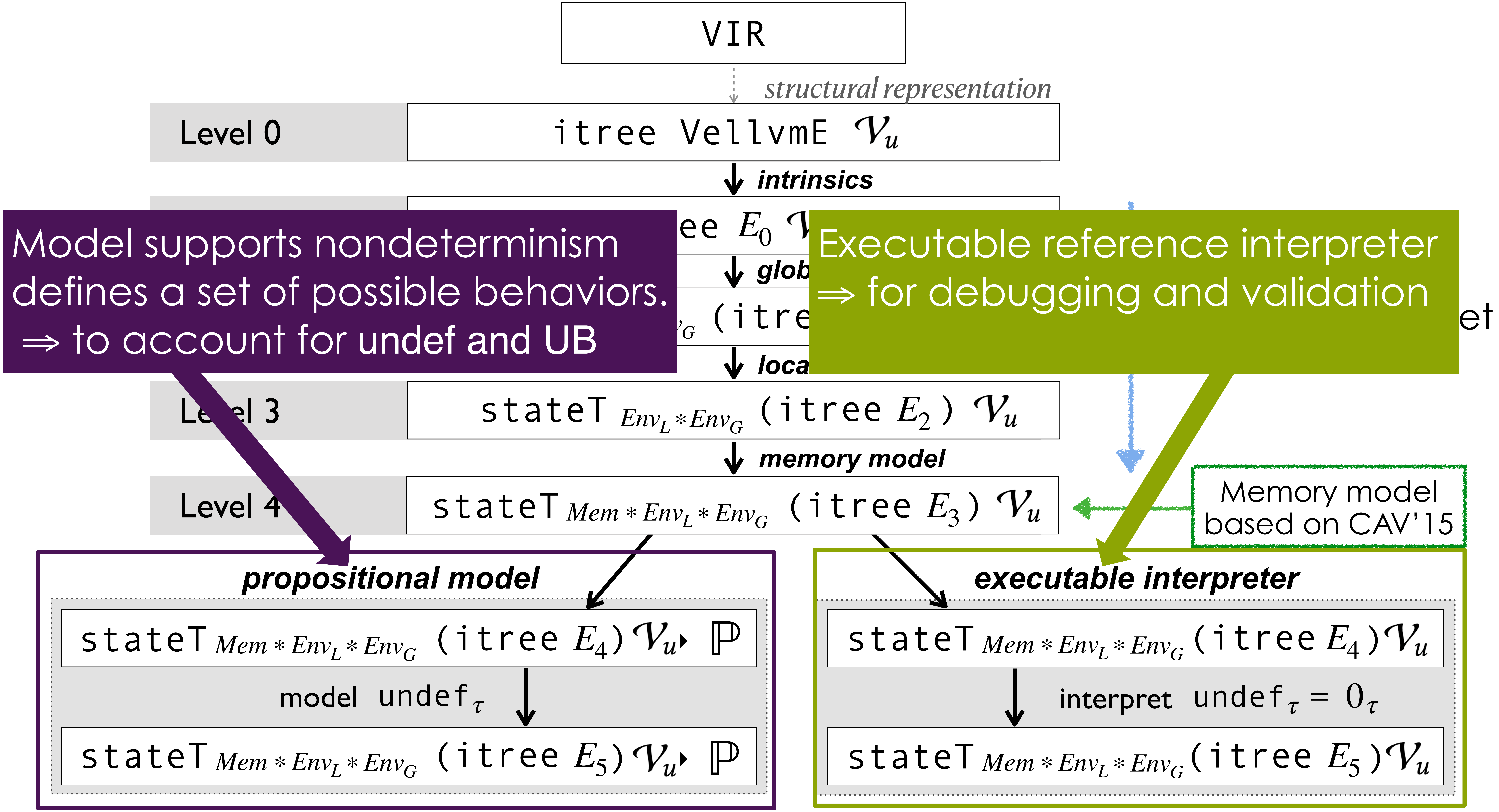

VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
Level 1
itree E_0 $\mathcal{V}_u$
global environment
Level 2
stateT$_{Env_G}$ (itree E_1) $\mathcal{V}_u$
Pieces of state get introduce


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) $\mathcal{V}_u$
Pieces of state get introduce

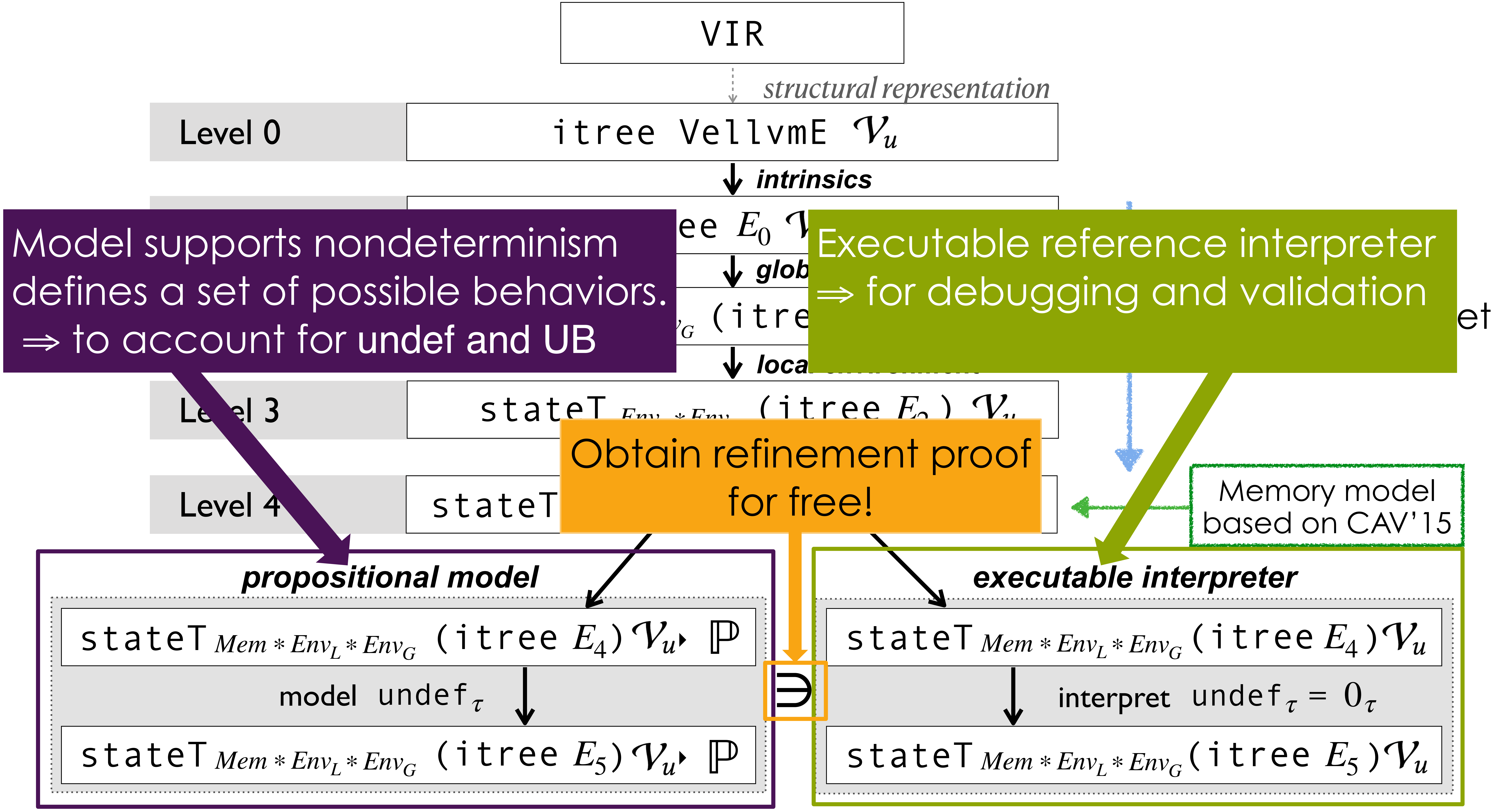


VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
Level 1
itree E_0 $\mathcal{V}_u$
global environment
Level 2
stateT$_{Env_G}$ (itree E_1) $\mathcal{V}_u$
local environment
Level 3
stateT $_{Env_L * Env_G}$ (itree E_2) $\mathcal{V}_u$
memory model
Level 4
stateT $_{Mem * Env_L * Env_G}$ (itree E_3) $\mathcal{V}_u$
Pieces of state get introduce
Memory model based on CAV'15


VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
ee E_0 $\mathcal{V}_u$
global environment
(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) $\mathcal{V}_u$
Model supports nondeterminism
defines a set of possible behaviors.
⇒ to account for undef and UB
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) $\mathcal{V}_u$, $\mathbb{P}$
Pieces of state get introduce
Memory model based on CAV'15



VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
ee E_0
glob
(itre
loca
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) $\mathcal{V}_u$
Model supports nondeterminism defines a set of possible behaviors.
⇒ to account for undef and UB
Executable reference interpreter
⇒ for debugging and validation
Memory model based on CAV'15
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) $\mathcal{V}_u$ $\mathbb{P}$
executable interpreter
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) $\mathcal{V}_u$


VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
Level 3
Level 4
Model supports nondeterminism defines a set of possible behaviors.
⇒ to account for undef and UB
Executable reference interpreter
⇒ for debugging and validation
Obtain refinement proof for free!
Memory model based on CAV'15
propositional model
stateT $Mem * Env_L * Env_G$ (itree E_4) $\mathcal{V}_u$, ℙ
model undef$_\tau$
stateT $Mem * Env_L * Env_G$ (itree E_5) $\mathcal{V}_u$, ℙ
⊇
executable interpreter
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) $\mathcal{V}_u$

But How Does it Relate to LLVM IR?

You can play with it yourself!

```
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

You can play with it yourself!

```
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 (OCaml) → Extracted interpreter → Tiny OCaml driver to crawl the tree (OCaml) → 120

Tiny OCaml driver to crawl the tree:

- External calls
- Debugging messages
- Failure

Realistic* (sequential) subset, happy to take feature requests!

* See the paper for the details of the features we cover

You can play with it yourself!

```
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 (OCaml) → Extracted interpreter → Tiny OCaml driver to crawl the tree (OCaml) → 120

External calls
Debugging messages
Failure

Realistic* (sequential) subset, happy to take feature requests!

Tested against ***clang*** over:

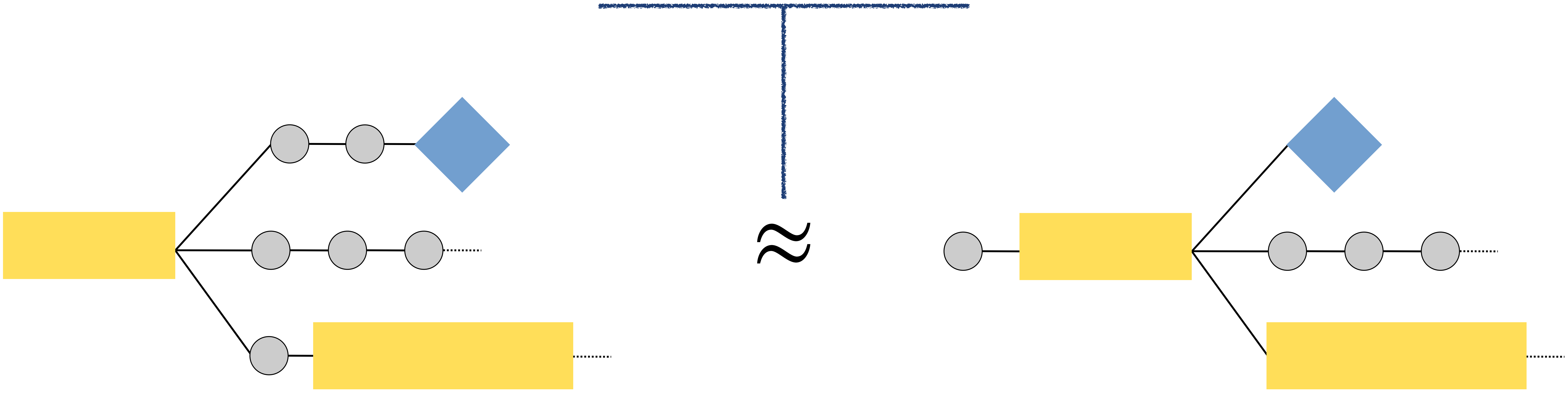
- A collection of unit tests
- A handful of significant programs from the HELIX project
- Early experiments over randomly generated programs using QuickChick

* See the paper for the details of the features we cover

But Why Would it Be Any Useful?

A (weak) bisimulation over itrees

Coinductive relation ignoring finite amounts of internal steps



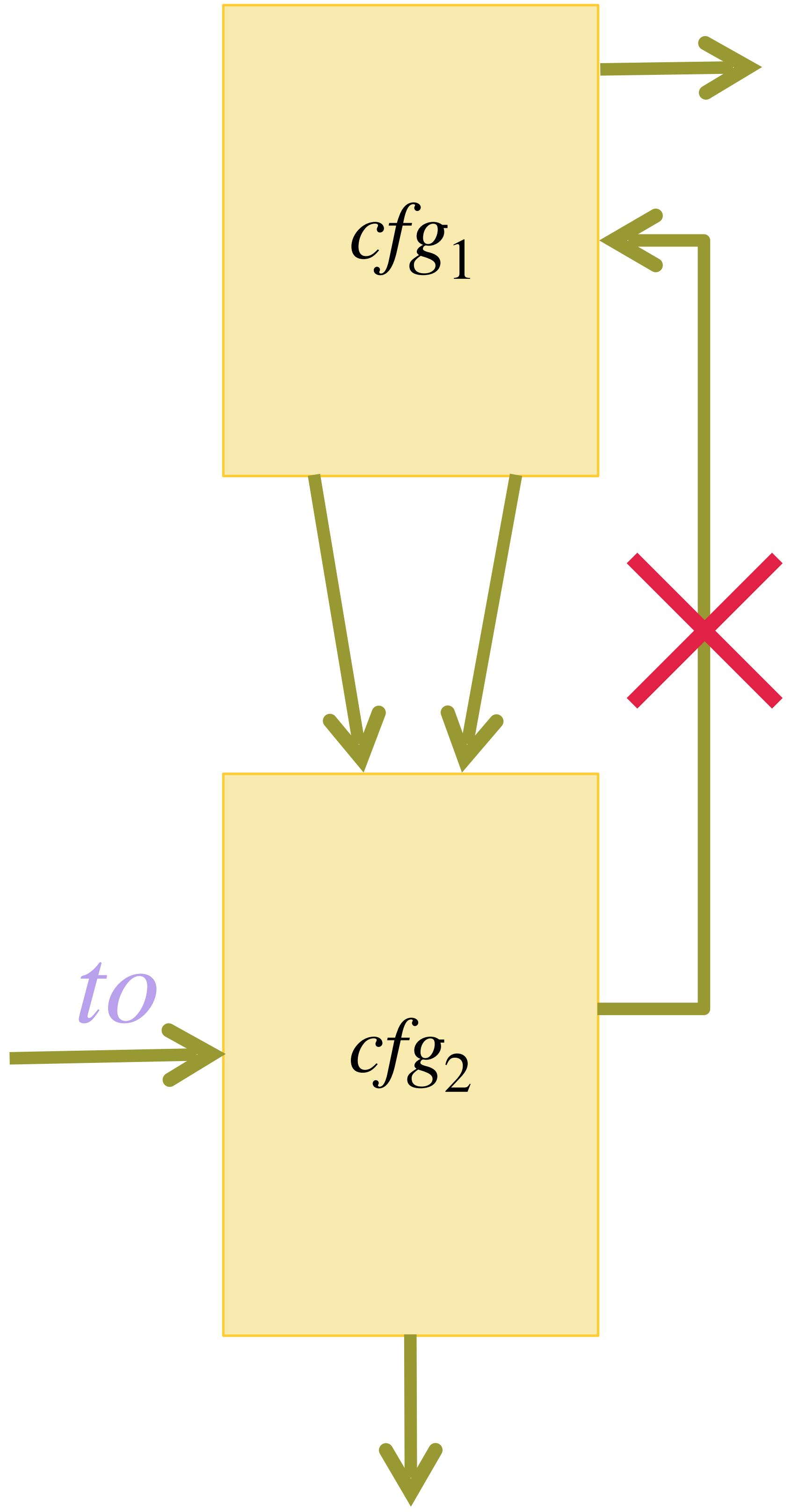
Get us a first (fine) notion of equivalent programs

Structural Equational Theory and Compositional Reasoning

A battery of structural equational lemmas at the VIR level

Structural Equational Theory and Compositional Reasoning

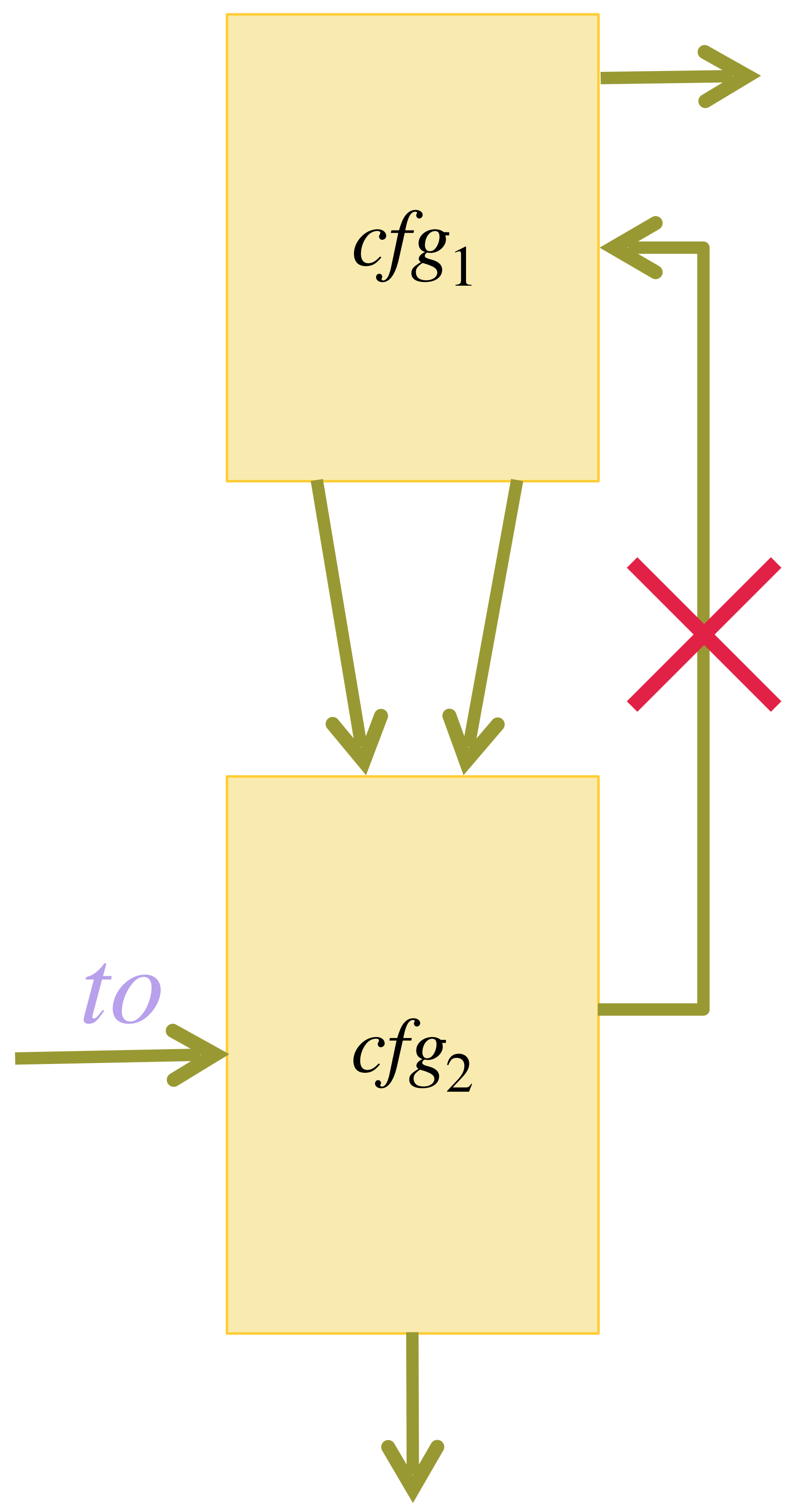
A battery of structural equational lemmas at the VIR level



Reasoning about control-flow graph composition

$$\frac{\mathbf{outputs}(cfg_2) \cap \mathbf{inputs}(cfg_1) = \emptyset \qquad to \notin \mathbf{inputs}(cfg_1)}{[\![cfg_1 \cup cfg_2]\!]\,(f, to) \approx [\![cfg_2]\!](f, to)}$$

Structural Equational Theory and Compositional Reasoning

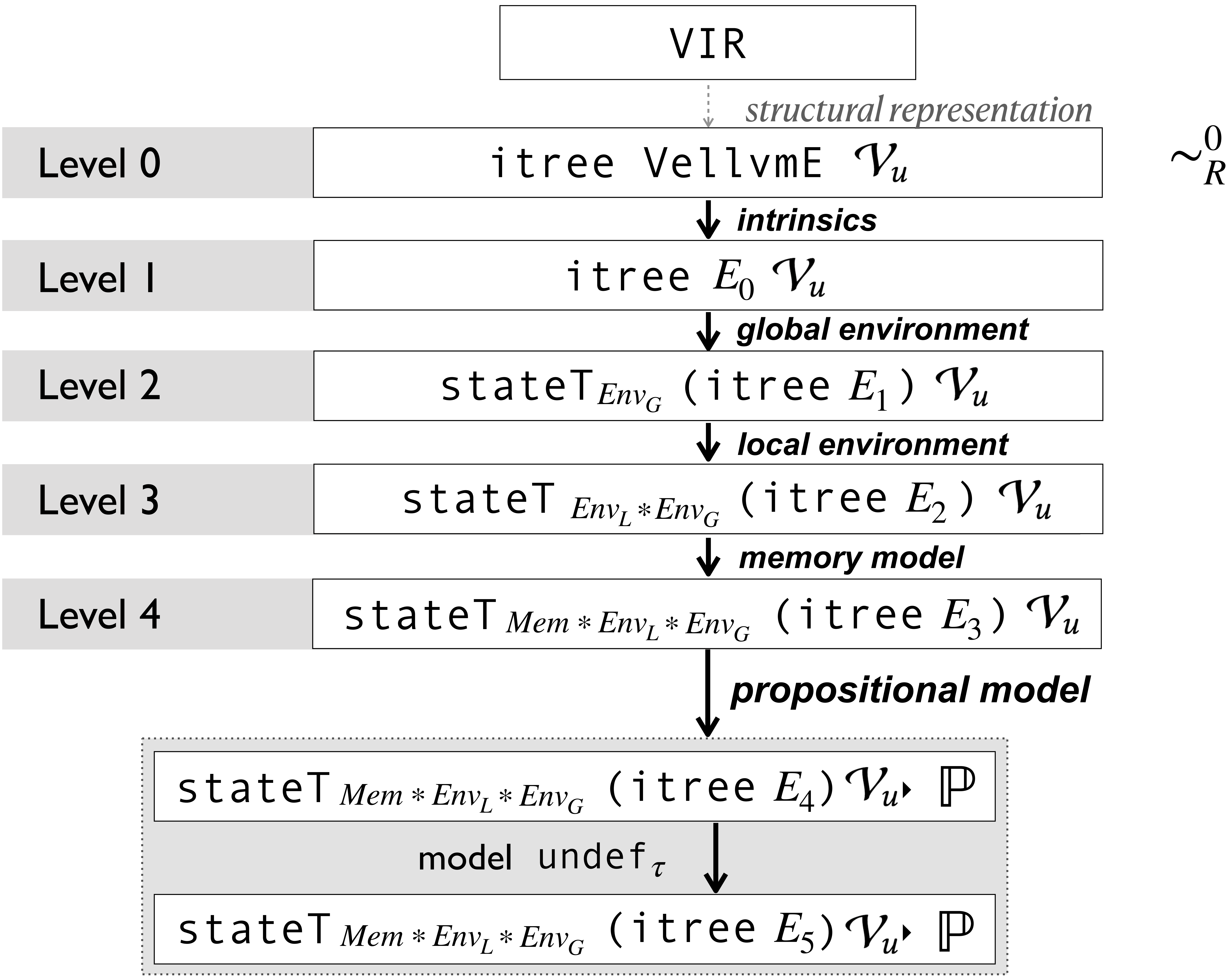


A battery of structural equational lemmas at the VIR level

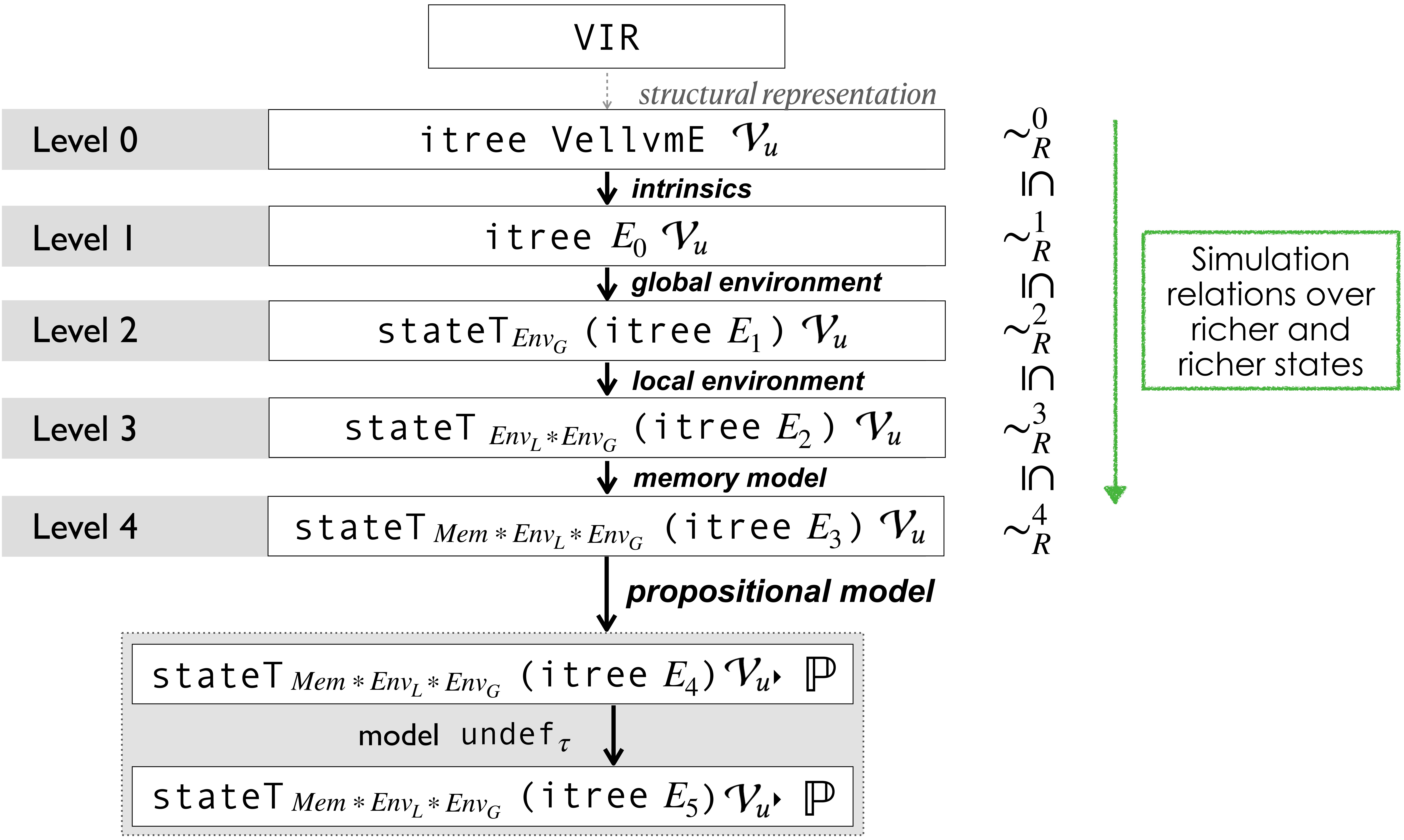
Reasoning about control-flow graph composition

$$\frac{\mathbf{outputs}(cfg_2) \cap \mathbf{inputs}(cfg_1) = \varnothing \qquad to \notin \mathbf{inputs}(cfg_1)}{[\![cfg_1 \cup cfg_2]\!]\,(f, to) \approx [\![cfg_2]\!](f, to)}$$

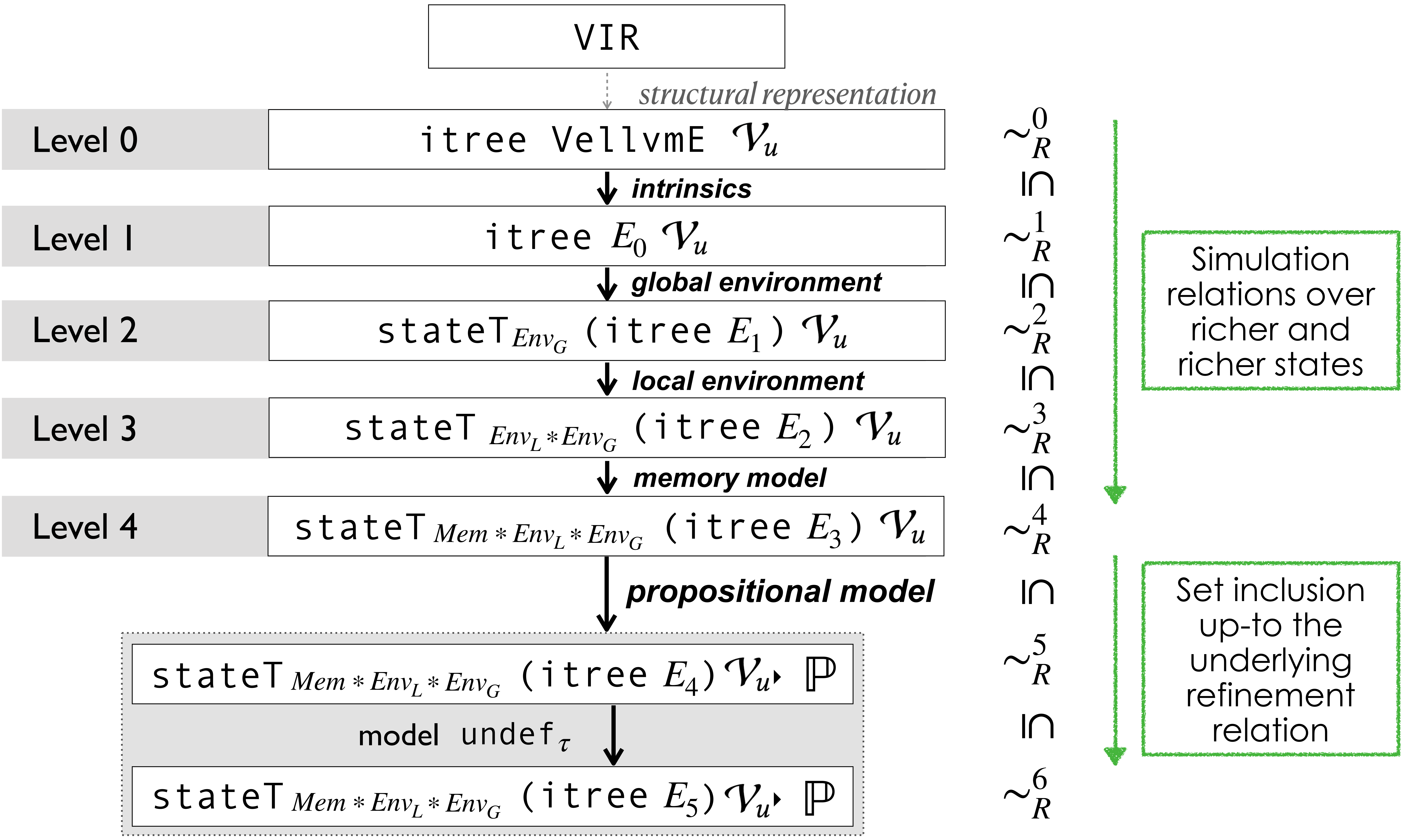
Proof of a simple *block-fusion optimization*

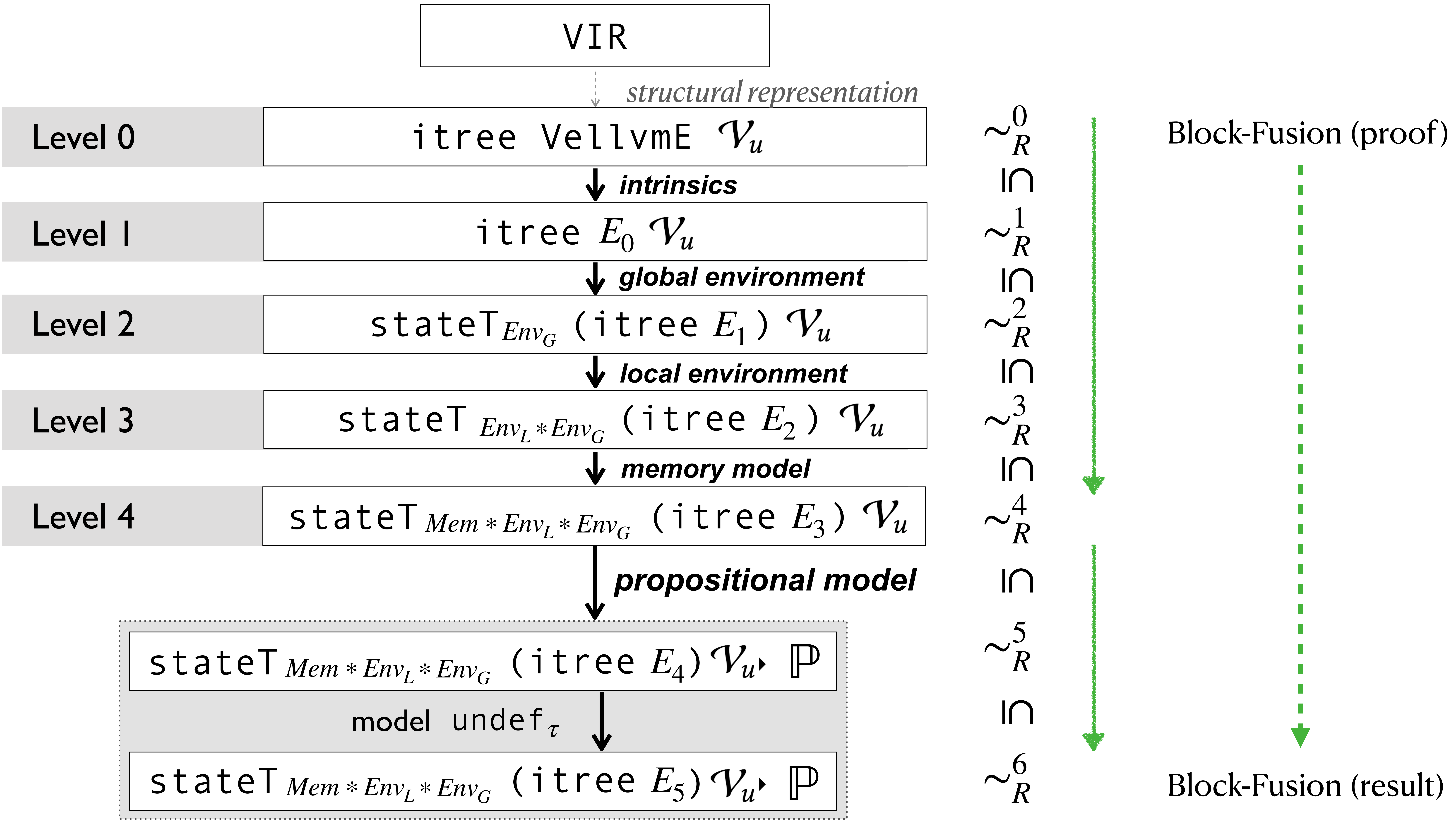


VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
$\sim_R^0$
intrinsics
Level 1
itree E_0 $\mathcal{V}_u$
global environment
Level 2
stateT$_{Env_G}$ (itree E_1) $\mathcal{V}_u$
local environment
Level 3
stateT$_{Env_L * Env_G}$ (itree E_2) $\mathcal{V}_u$
memory model
Level 4
stateT$_{Mem * Env_L * Env_G}$ (itree E_3) $\mathcal{V}_u$
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) $\mathcal{V}_u$ $\mathbb{P}$



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) $\mathcal{V}_u$
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) $\mathcal{V}_u$ $\mathbb{P}$
$\sim_R^0 \supseteq \sim_R^1 \supseteq \sim_R^2 \supseteq \sim_R^3 \supseteq \sim_R^4$
Simulation relations over richer and richer states


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) $\mathcal{V}_u$
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) $\mathcal{V}_{u'}$ $\mathbb{P}$
$\sim_R^0$ ⊇ $\sim_R^1$ ⊇ $\sim_R^2$ ⊇ $\sim_R^3$ ⊇ $\sim_R^4$ ⊇ $\sim_R^5$ ⊇ $\sim_R^6$
Simulation relations over richer and richer states
Set inclusion up-to the underlying refinement relation

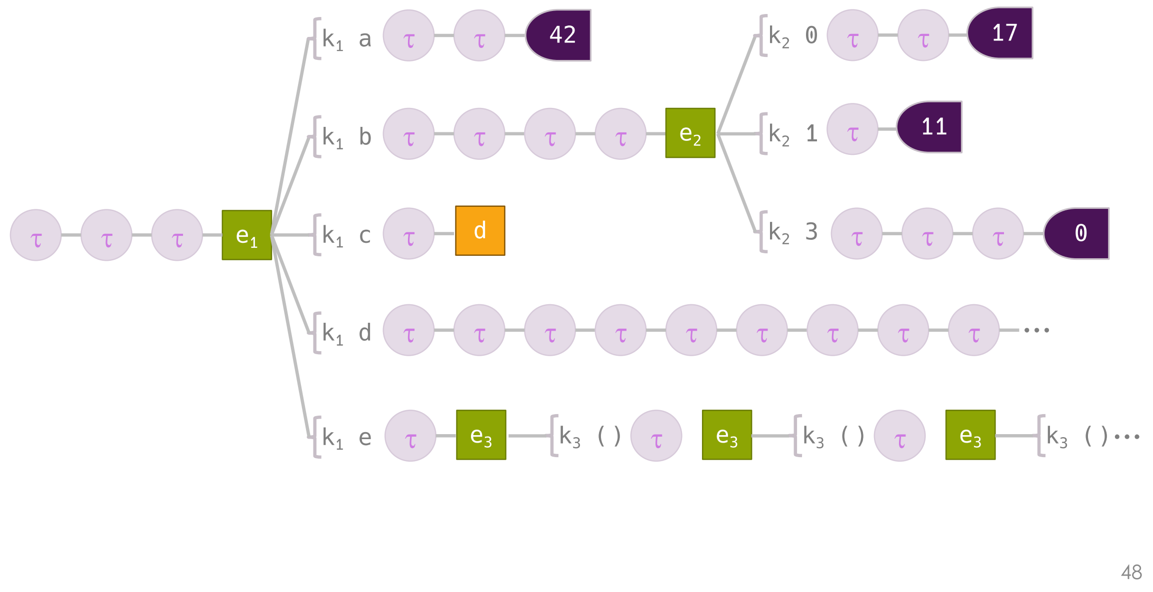


VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
Level 1
itree E_0 $\mathcal{V}_u$
global environment
Level 2
stateT$_{Env_G}$ (itree E_1) $\mathcal{V}_u$
local environment
Level 3
stateT $_{Env_L * Env_G}$ (itree E_2) $\mathcal{V}_u$
memory model
Level 4
stateT $_{Mem * Env_L * Env_G}$ (itree E_3) $\mathcal{V}_u$
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) $\mathcal{V}_u$ $\mathbb{P}$
$\sim_R^0$ ⊆ $\sim_R^1$ ⊆ $\sim_R^2$ ⊆ $\sim_R^3$ ⊆ $\sim_R^4$ ⊆ $\sim_R^5$ ⊆ $\sim_R^6$
Block-Fusion (proof)
Block-Fusion (result)

Instruction level reasoning

To reason about instructions, we could get back down to comparing trees

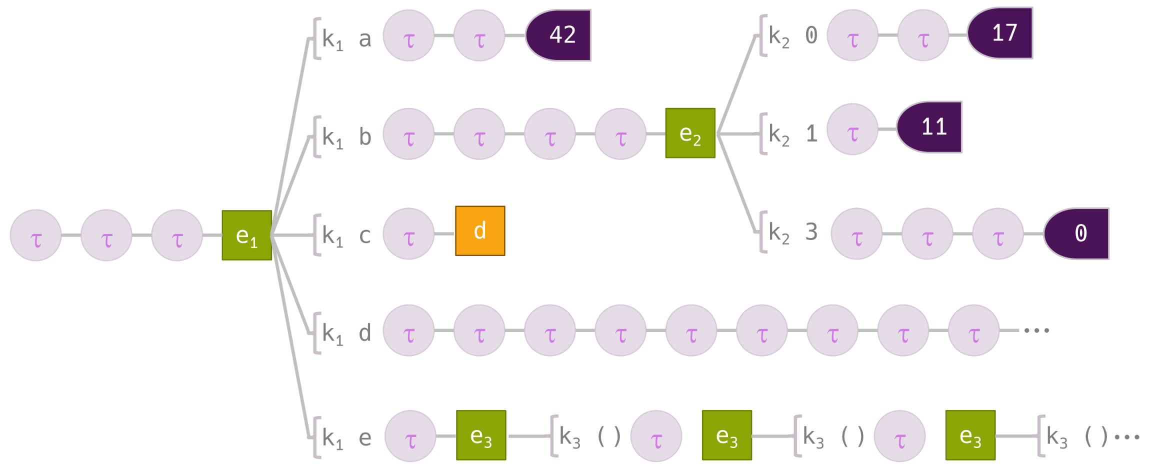
⟦%x = load i64, i64* %acc⟧$_{instr}$ →



Instruction level reasoning

To reason about instructions, we could get back down to comparing trees

$\llbracket \%x = \text{load i64, i64* } \%acc \rrbracket_{instr}$ ⟶



Instead, we reason at the level of VIR through a battery of lemmas for each expression and instruction

$$\frac{\llbracket acc \rrbracket_{expr}\ g\ l\ m \approx \text{Ret}\ (m, (l', (g, tt))) \qquad \text{read}\ m\ a\ \text{i64} = \text{inr}\ uv}{\llbracket \%x = \text{load i64, i64* } \%acc \rrbracket_{instr}\ g\ l\ m \approx \text{Ret}\ (m, (\text{Maps.add}\ x\ uv\ l', (g, tt)))}$$

Representation functions can be made completely opaque

Two main reasoning ingredients

Strong equivalences at the VIR level over:

- the syntactic structure of the language
- the control flow
- the instructions, expressions and terminators.

Symbolic interpreter that can be run by rewriting during refinement proofs

Two main reasoning ingredients

Strong equivalences at the VIR level over:

- the syntactic structure of the language
- the control flow
- the instructions, expressions and terminators.

Symbolic interpreter that can be run by rewriting during refinement proofs

A primitive relational program logic:

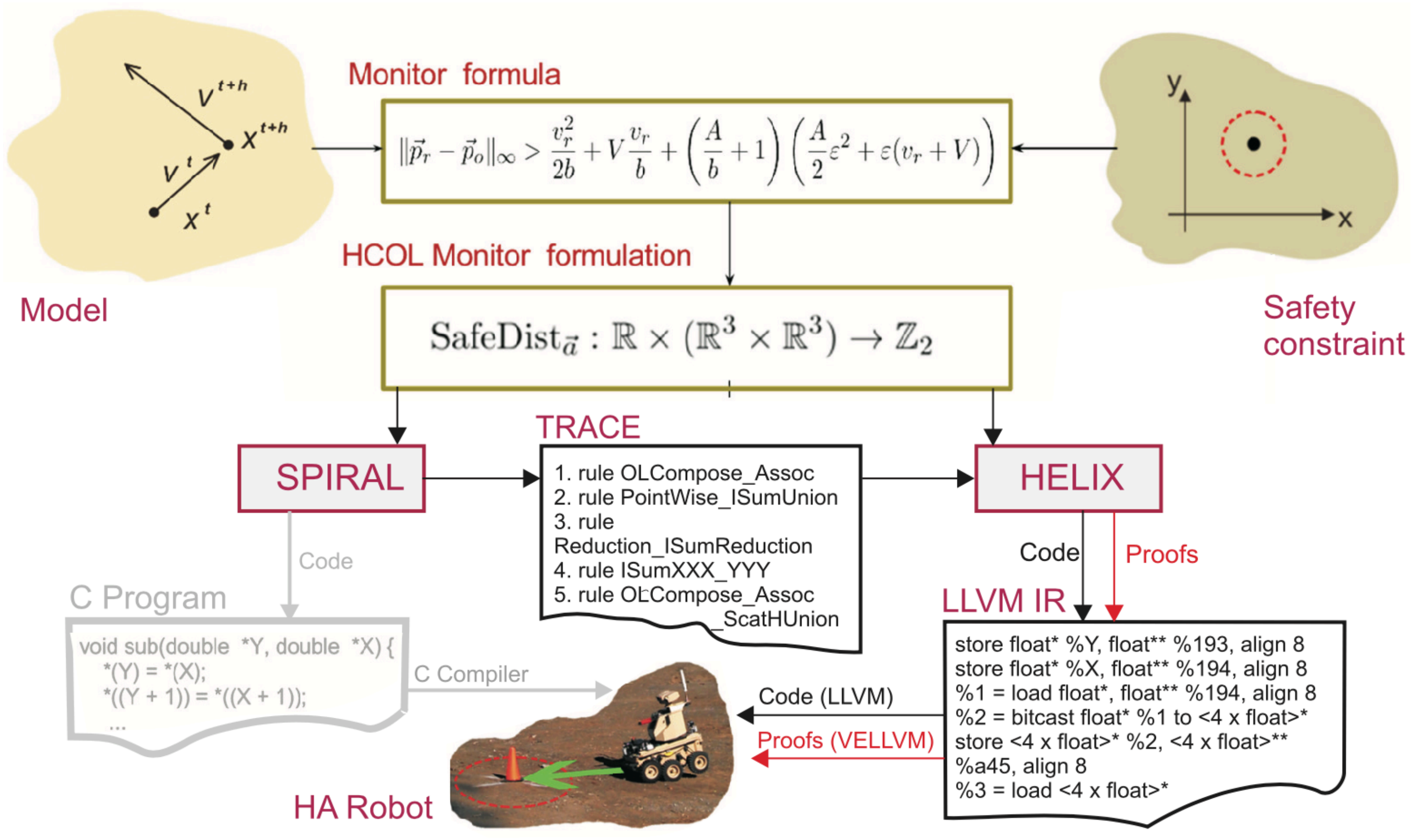
- Weakening, conjunction, … over the postcondition
- Sequential composition

Compositional construction of refinement proofs

SPIRAL/HELIX

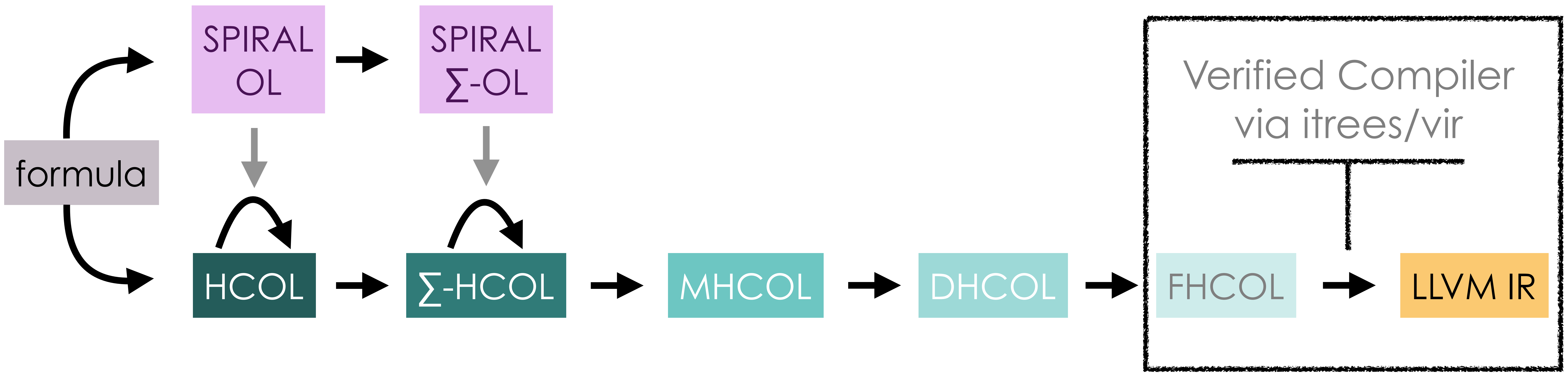
[Püschel, et al. 2005] [Franchetti et al., 2005, 2018] [Zaliva et al., 2015 2018, 2019]

DSL for high-performance numerical computing.



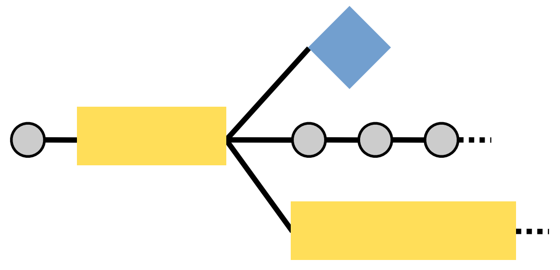
- Numerical computations compiled down to LLVM IR
- Formalized in Coq, targets Vellvm
- Bottom of the compilation chain proved* w.r.t. this technique

* Some operators are currently not proved



Vellvm is Back!

Interaction Trees (itrees)



github.com/DeepSpec/InteractionTrees

opam install coq-itree

Used to build

(Re) Vellvm

github.com/vellvm/vellvm

Soon(ish) in opam!

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

- a certified interpreter
- promising modularity
- a rich equational theory
- an equational style to refinement proofs

A fertile ground is laid!

Verified analyses

Verified optimizations

Concurrency

Back-ends