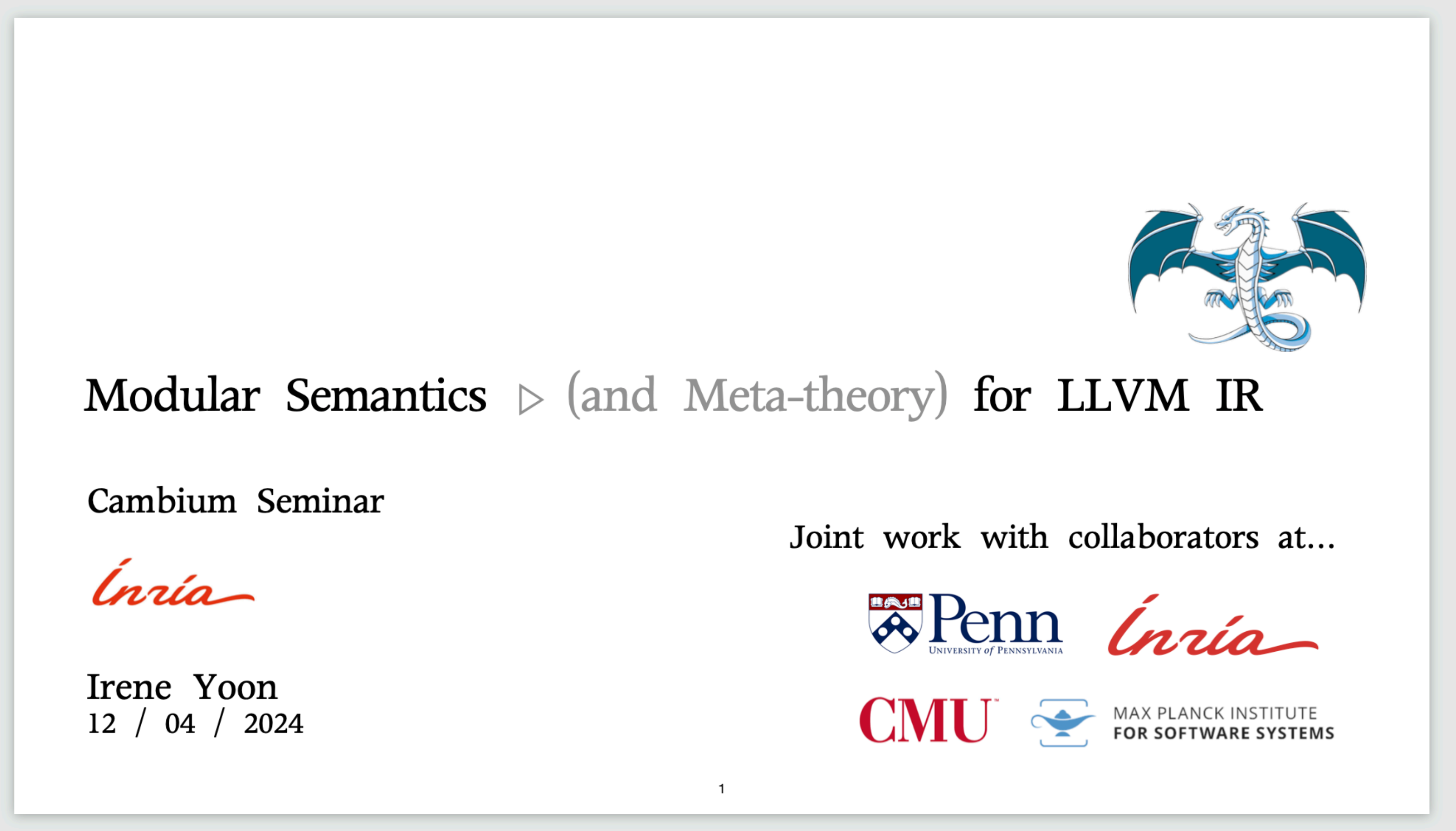


Modular Semantics ▷ (and Meta-theory) for LLVM IR

Cambium Seminar

Irene Yoon
12 / 04 / 2024

Joint work with collaborators at...

Velliris

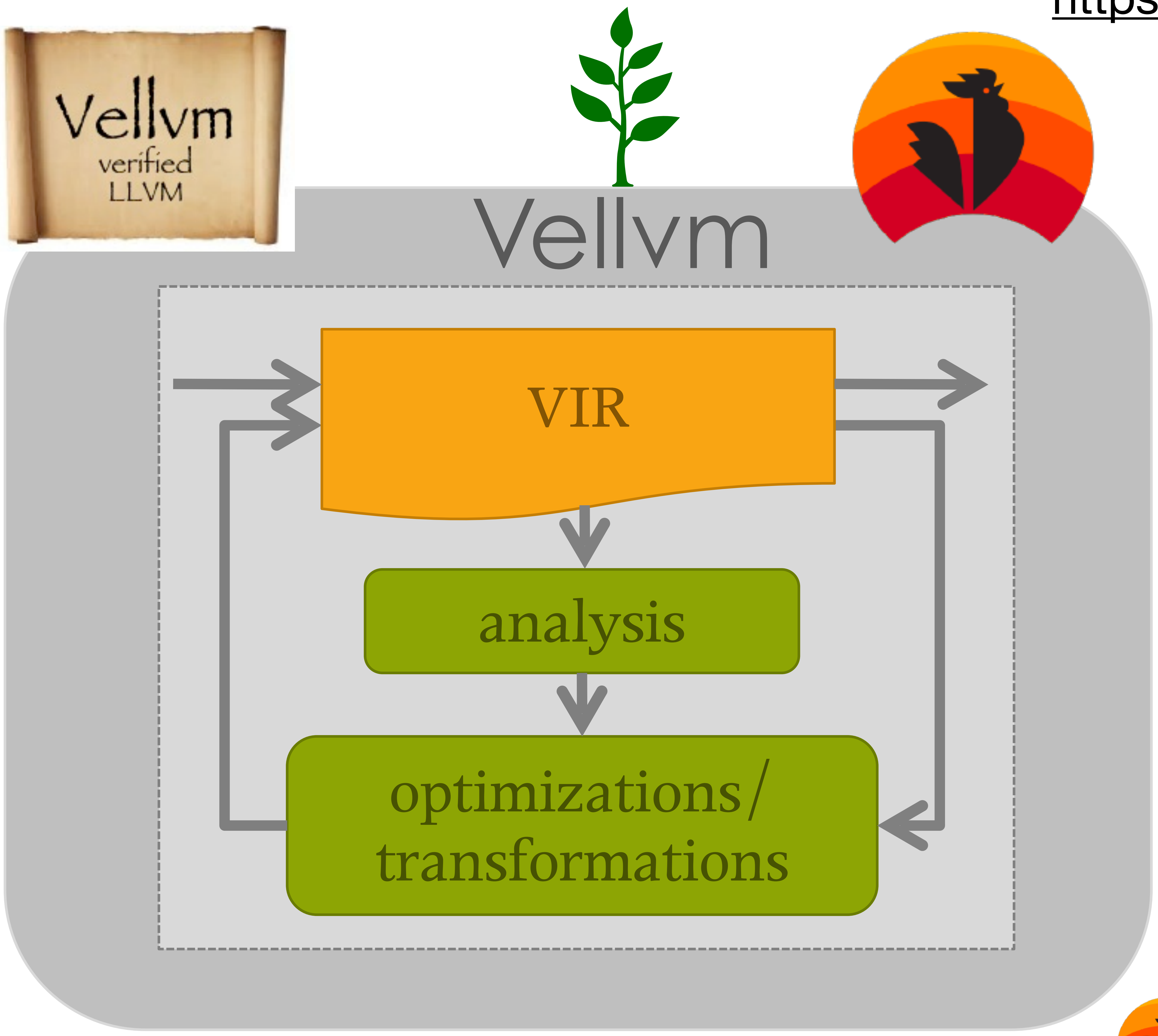
A Relational Separation Logic for LLVM IR

Irene Yoon, Simon Spies, Youngju Song, Lennard Gäher, Derek Dreyer, Steve Zdancewic

Irene Yoon | Cambium Seminar | 24/05/24

Vellvm 2.0 Overview

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. - ICFP 2024 (conditionally accepted)]
Infinite/finite memory model for LLVM IR

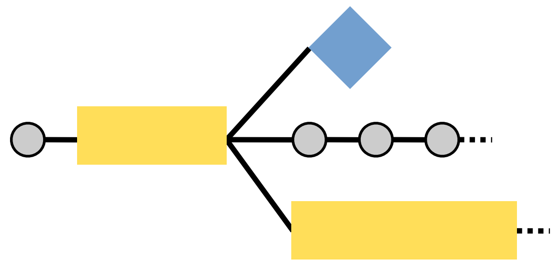
[Yoon et al. - Today's talk]
Relational separation logic for LLVM IR

* : all results mechanized in the Coq Proof Assistant

Vellvm 2.0

Interaction Tree based semantics for LLVM IR

Interaction Trees (itrees)



[Xia et al. 2020]

github.com/DeepSpec/InteractionTrees

Used to build

(Re)Vellvm

github.com/vellvm/vellvm

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

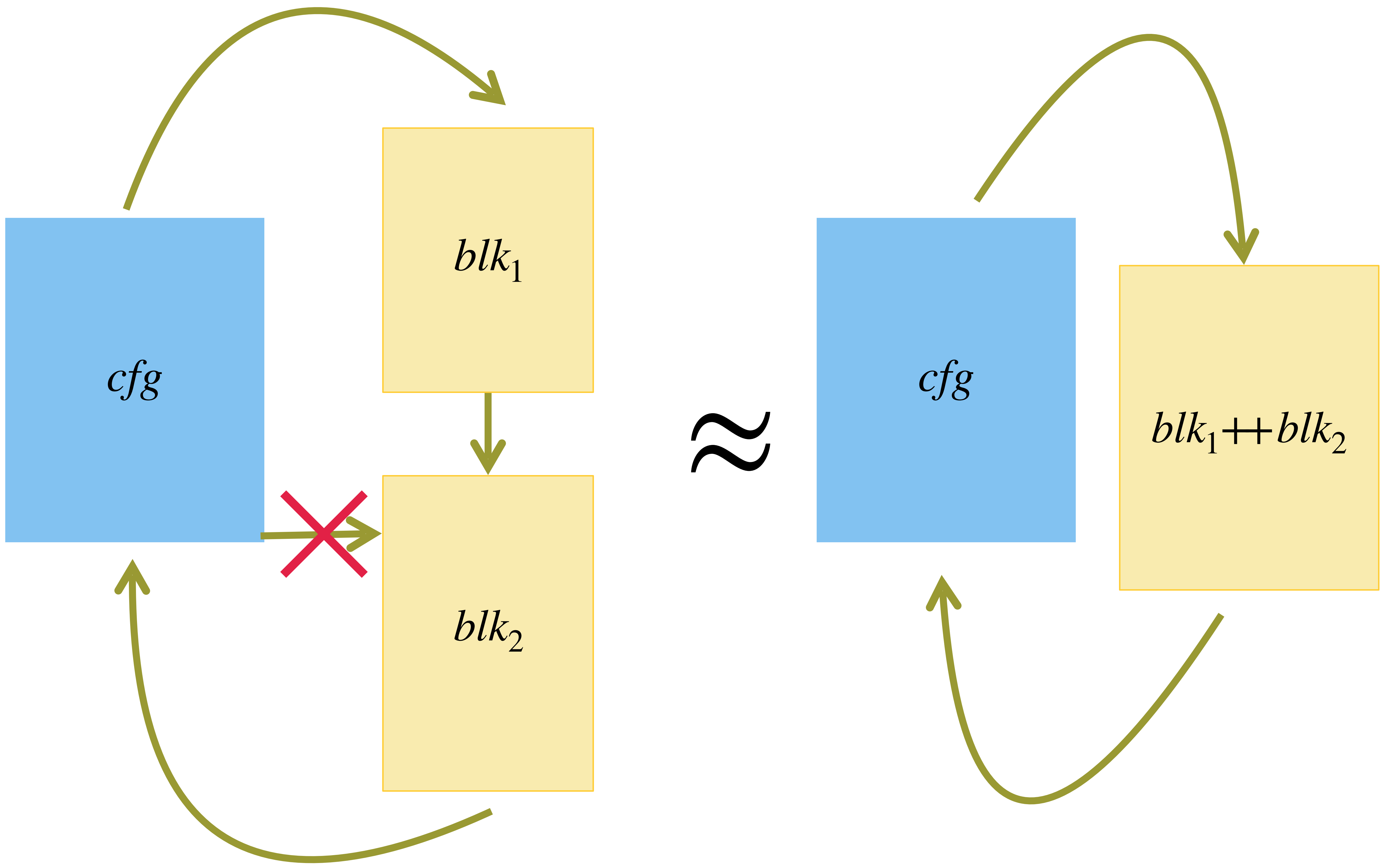
Semantics: Compositional, Modular, Executable

Reasoning: Equational, termination sensitive

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

Benefits of ITree-based reasoning

Reasoning about control flow



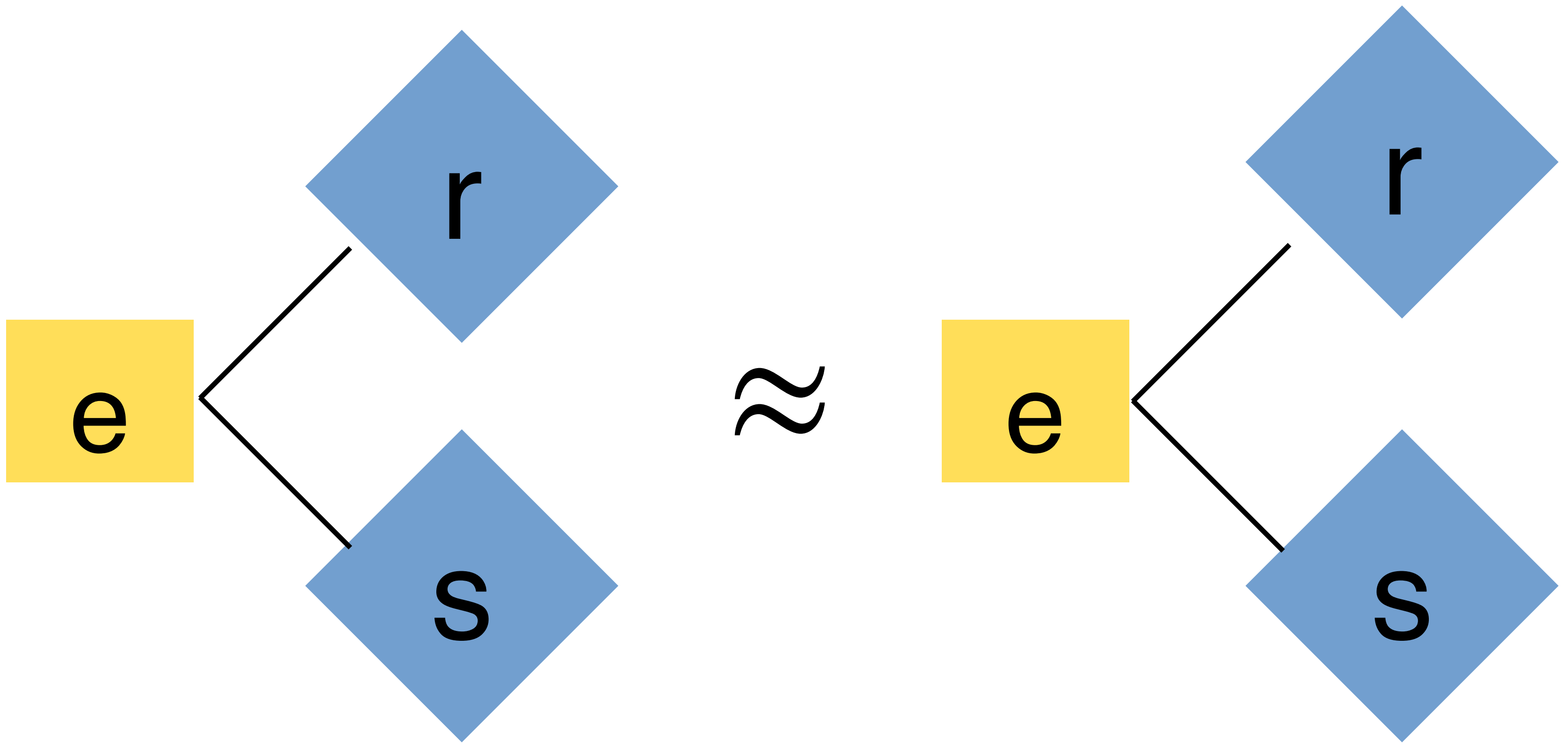
- Proof of block-merging optimization
- Reasoning about control flow is simple (if it does not change the trace of events)

Weak bisimulation on Interaction Trees

Termination-sensitive relational reasoning

$$e_t \approx_R e_s$$

"eutt"



- Program e_t and e_s are related to each other (i.e. bisimilar) w.r.t. a value relation R
  - If one program diverges, the other must diverge in a similar way (& vice versa.)
  - If one program terminates, the other must terminate in a similar way (& vice versa.) and the returned values are related by the value relation R

Benton-style relational Hoare reasoning


Let's reason about programs via Hoare triples (quadruple)!

- "eutt" can be seen as a partial (e.g. without a proof of termination) relational Hoare triple with a trivial precondition

$$e_t \approx_R e_s \qquad\qquad \{\ \top\ \}\ e_t \approx e_s\ \{\ R\ \}$$

- A partial Hoare triple can be instantiated by taking the diagonal:

$$\{\ \top\ \}\ e\ \{Q\} := e \approx_{\lambda x,y.\ x=y\ \wedge\ Q\ x}\ e$$

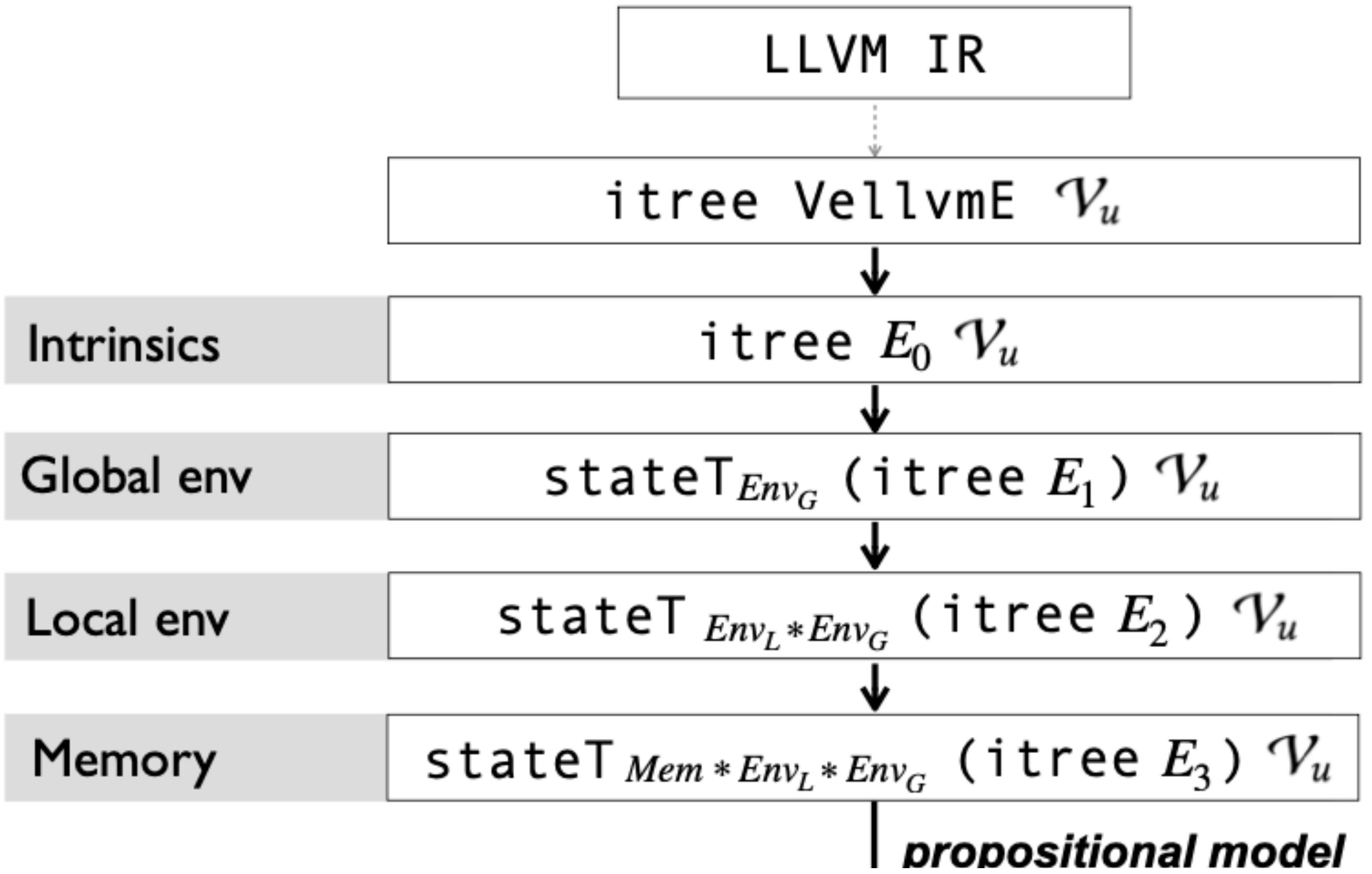
Benton-style relational Hoare reasoning

Let's reason about programs via Hoare triples (quadruple)!

- There are not many transformations that preserve interactions with state

- After one step of interpretation (of a state with carrier type S) ⟦ - ⟧, we see a state-passing tree (S -> itree F (S * A)). The post-condition on eutt relates both values and the final state.

$$[\![e_t]\!] \sigma_t \approx_Q [\![e_s]\!] \sigma_s$$



`LLVM IR`

$\texttt{itree VellvmE } \mathcal{V}_u$

Intrinsics $\texttt{itree } E_0 \ \mathcal{V}_u$

Global env $\texttt{stateT}_{Env_G} \ (\texttt{itree } E_1) \ \mathcal{V}_u$

Local env $\texttt{stateT}_{Env_L * Env_G} \ (\texttt{itree } E_2) \ \mathcal{V}_u$

Memory $\texttt{stateT}_{Mem * Env_L * Env_G} \ (\texttt{itree } E_3) \ \mathcal{V}_u$

propositional model

- We can have relational pre- and post-conditions about the initial and final states of programs

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

Global state invariants and assumptions

A need for modularity

- The pre- and post- condition must carry global invariants about the state

Given a stateful interpretation function $[\![-]\!]$,

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

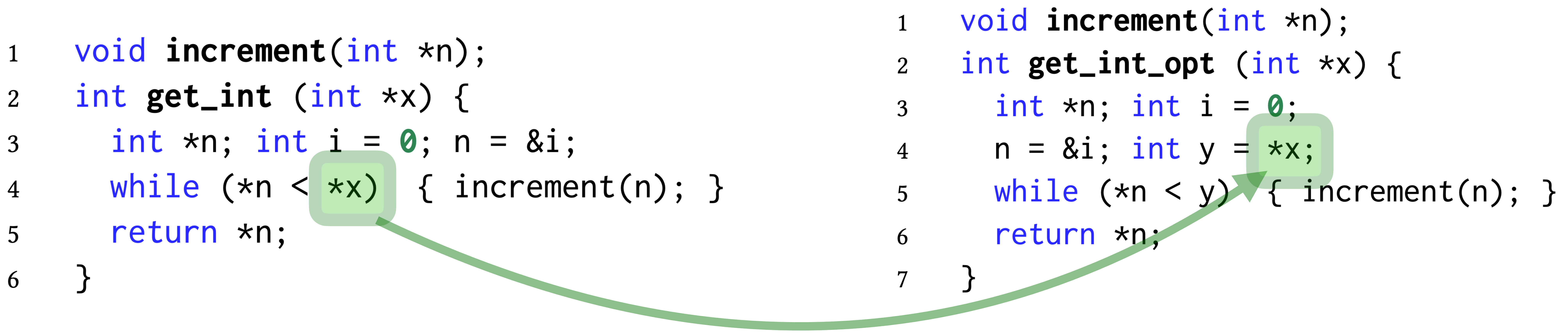
… and global invariants are difficult to work with!

In particular, in the setting of LLVM IR..

1. LLVM IR transformations make assumptions over memory regions that call for localized reasoning
2. Stack-allocated regions allocated using "alloca" are automatically collected upon function return, and local variables live in the scope of a function: it would be nice for the logic to be aware of this stack discipline

Example: Loop invariant code motion

What invariants does the compiler assume for its optimizations?



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

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

- LLVM optimizations (1) reorder (or modify/remove) memory-related instructions, and (2) often make certain assumptions about external calls while doing so
- By adding an annotation at the generated LLVM IR (function attribute) for the C code above, one can specify that the function only accesses memory through its arguments

```
declare i32 @increment(i32*) argmemonly
```

> ” function can only affect memory accessible by the arguments passed on to the function ”

Another example: Load-after-store on "promotable" locations

Compilers want to use assumptions from static analysis passes

- LLVM IR transformation often uses assumptions derived from analysis passes (e.g. alias analysis), and from the perspective of verifying optimizations, we need a way to state these assumptions

```
%a = alloca i32
...
store i32 6, i32 %a
...
%b = load %a
ret %b
...
```

```
%a = alloca i32
...
store i32 6, i32 %a
...
ret 6
...
```

- ″promotable″ register: no aliasing, no storing to memory

Separation Logic

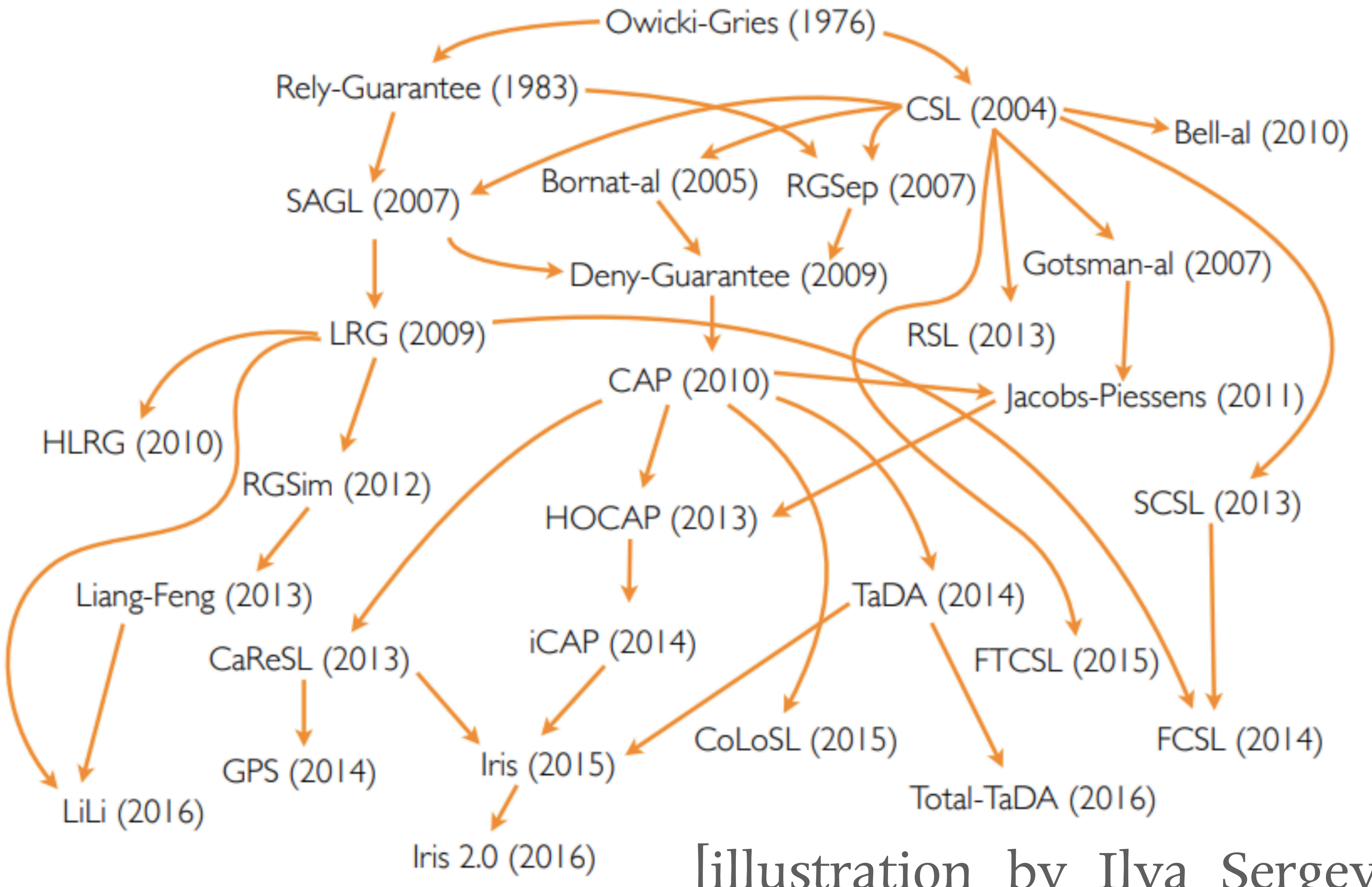
Localized reasoning about resources for all!

- Separation logic [O. Hearn et al.]

$$P * Q \qquad \frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

- Iris [Jung et al.]: a higher-order concurrent separation logic framework
  - Highly reusable and influential in consolidating variants of separation logics
  - Used for various other realistic semantics (RustBelt, RefinedC, Iris-WASM, etc).

The genealogy of separation logics



[illustration by Ilya Sergey]

Can we bring the niceties of separation logic to LLVM IR?

<u>Semantics*</u> <u>Program logic</u>

VIR + Iris = Velliris

Relational separation logic for LLVM IR!

[Gäher et al.]

Simuliris : relational Hoare logic in Iris

- Focusing rules on source and target programs
- Termination-sensitive simulations in Iris

$$\begin{tikzcd}
 & \{P\}\ e_t \preceq e_s\ \{v_t, v_s. Q\} \arrow[dl, bend right, "\text{focus source}"'] \arrow[dr, bend left, "\text{focus target}"] & \\
\{P\}\ e_s\ \{e_s'.R\}^{\mathrm{src}} & & \{P\}\ e_t\ \{e_t'.R\}^{\mathrm{tgt}}
\end{tikzcd}$$

$$\dfrac{\{P\}\ e_s\ \{v_s.\ \Psi\ v_s\}^{\mathrm{src}} \qquad \forall v_s.\ \Psi\ v_s \mathrel{-\!\!*}\ e_t \preceq k_s\ v_s\ \{\Phi\}}{\{P\}\ e_t \preceq x \leftarrow e_s\ ;;\ k_s\ x\ \{\Phi\}}\ \textsc{SourceFocus}$$

$$\dfrac{\{P\}\ e_t\ \{v_t.\ \Psi\ v_t\}^{\mathrm{tgt}} \qquad \forall v_t.\ \Psi\ v_t \mathrel{-\!\!*}\ k_t\ v_t \preceq e_s\ \{\Phi\}}{\{P\}\ x \leftarrow e_t\ ;;\ k_t\ x \preceq e_s\ \{\Phi\}}\ \textsc{TargetFocus}$$

$$\dfrac{\{P\}\ e_t \preceq e_s\ \{v_t\ v_s.\ \Psi\ v_t\ v_s\} \qquad \forall v_t\ v_s.\ \Psi\ v_t\ v_s \mathrel{-\!\!*} k_t\ v_t \preceq k_s\ v_s\ \{\Phi\}}{\{P\}\ (x \leftarrow e_t\ ;;\ k_t\ x) \preceq (x \leftarrow e_s\ ;;\ k_s\ x)\ \{\Phi\}}\ \textsc{SimBind}$$

$$\{\Phi\ v_t\ v_s\}\ \mathrm{Ret}\ v_t \preceq \mathrm{Ret}\ v_s\ \{\Phi\}\ \textsc{SimValue} \qquad \dfrac{\{P\}\ e_t \preceq e_s\ \{\Phi\}}{\{P * R\}\ e_t \preceq e_s \{v_t\ v_s.\ \Phi\ v_t\ v_s * R\}}\ \textsc{SimFrame}$$

(Typical) Recipe to use Iris

(1) Ingredient: an abstract view on state (ghost theory) using separation logic resources

Iris has a notion of <u>resource algebras</u> and generic constructions of resource algebras suitable for read-only map, permission-based ownership, etc.

For expository purposes, an example of a much simpler resource algebra.
Given a partial commutative monoid $(R, (\odot), \epsilon)$ we can define a ghost theory :

?

Given a heap which is a partial map from addresses to integers, we can define an ownership predicate $\ell \longmapsto v$ where

$(\ell \longmapsto v) \odot (\ell' \longmapsto w) := (\ell \longmapsto v; \ell' \longmapsto w)$ *where* $\ell \neq \ell'$

$(\ell \longmapsto v) \odot (\ell \longmapsto w) := \bot$

$L \odot L' := (L \mathbin{+\!+} L')$ *where* $\mathrm{dom}(L) \cap \mathrm{dom}(L') = \emptyset$

$L \odot L' := \bot$ *where* $\mathrm{dom}(L) \cap \mathrm{dom}(L') \neq \emptyset$

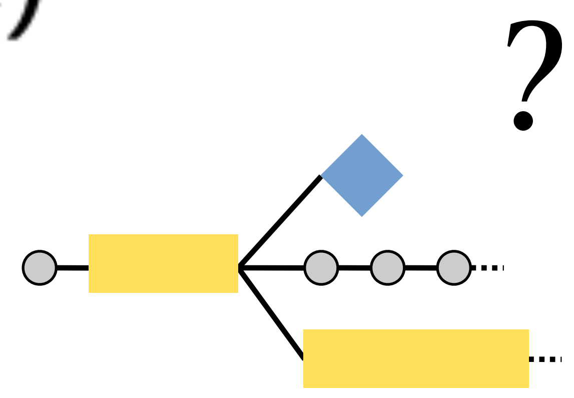
(Typical) Recipe to use Iris

(2) Ingredient: a small-step semantics

Given the small-step semantics and ghost theory, a Hoare triple can be derived via the typical weakest precondition model of Iris

$$\{P\}e\{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\{\Phi\})$$

Given a postcondition Φ, $\mathsf{wp}\ e\{\Phi\}$ gives the *weakest precondition* under which all executions of e are *safe* (i.e. does not get stuck) and all return values v satisfy $\Phi(v)$

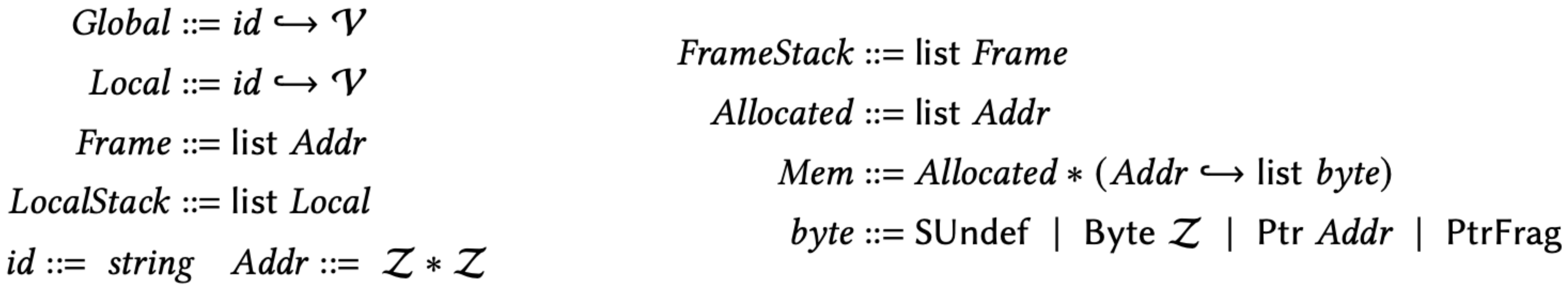


Roadmap

1. A taste of the stack-based ghost theory for LLVM IR
2. Memory-relevant attributes in LLVM IR
3. Model : A relational weakest-precondition model for Interaction Trees in Iris
4. Adequacy

VIR State

- The state of VIR: *Global * (Local * LocalStack) * (Mem * FrameStack)*



$$
\begin{aligned}
Global &::= id \hookrightarrow \mathcal{V} \\
Local &::= id \hookrightarrow \mathcal{V} \\
Frame &::= \mathsf{list}\ Addr \\
LocalStack &::= \mathsf{list}\ Local \\
id ::= \ string \quad Addr &::= \ \mathcal{Z} * \mathcal{Z}
\end{aligned}
$$

$$
\begin{aligned}
FrameStack &::= \mathsf{list}\ Frame \\
Allocated &::= \mathsf{list}\ Addr \\
Mem &::= Allocated * (Addr \hookrightarrow \mathsf{list}\ byte) \\
byte &::= \mathsf{SUndef} \ \mid\ \mathsf{Byte}\ \mathcal{Z} \ \mid\ \mathsf{Ptr}\ Addr \ \mid\ \mathsf{PtrFrag}
\end{aligned}
$$

- The stack resources need to managed upon function entry and exit
- The ghost theory in Velliris deals with this deallocation by keeping track of the stack frame and the associated set of stack-allocated locations.

Reasoning about local environments

Stack frame rules (not to be confused with the frame rule)

Ghost resources

$\mathsf{Frame}^{\mathsf{src}}\ i$	We are at index "i" on the stack frame.
$\langle \mathsf{id} := v \rangle_i^{\mathsf{src}}$	At frame index "i", we have access to local id "id" with value "v" stored on it.
$\mathsf{Local}_i^{\mathsf{src}}\ L$	At frame index "i", "L" is the domain of the local environment.

(Source side) Hoare triples

$$\frac{\{P\}\ e_s\ \{v_s.\ \Psi\ v_s\}^{\mathsf{src}} \qquad \forall v_s.\ \Psi\ v_s \mathrel{-\!\!*}\ e_t \preceq k_s\ v_s\ \{\Phi\}}{\{P\}\ e_t \preceq x \leftarrow e_s \,;;\ k_s\ x\ \{\Phi\}}\ \textsc{SourceFocus}$$

$\textsc{LocalRead}$

$$\begin{array}{l} \{\langle \mathsf{id} := v \rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i\} \\ \quad \mathtt{trigger}\ (\mathsf{LRd}^{\mathcal{V}_u}\ (\uparrow id)) \\ \{v_s'.\ v_s' = v * \langle \mathsf{id} := v \rangle_i^{\mathsf{src}}\}^{\mathsf{src}} \end{array}$$

We are currently at stack frame "i", and we know that the local environment stores "v" for "id".

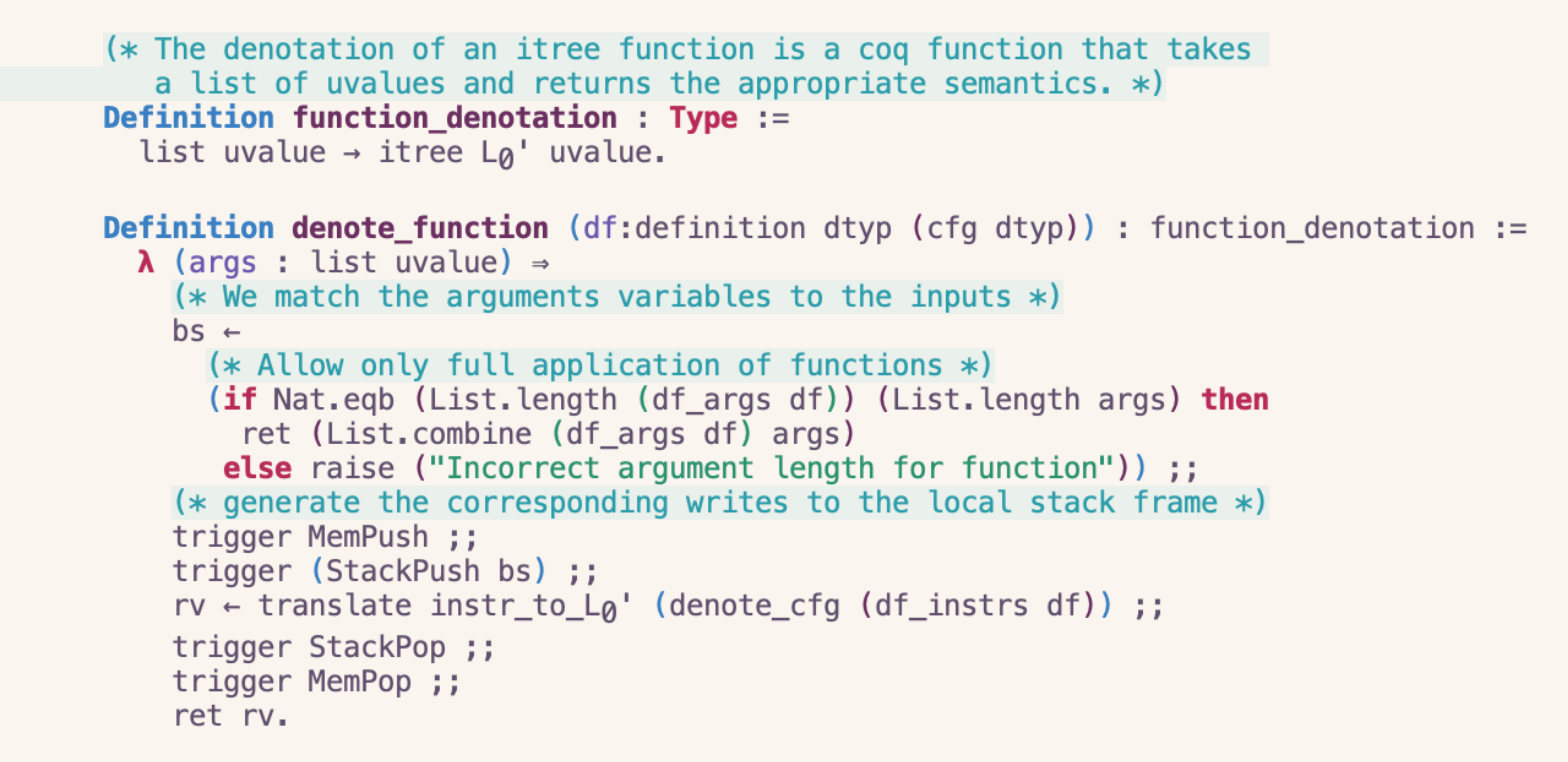
$\textsc{LocalWrite}$

$$\begin{array}{l} \{(id \notin L) * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ L\} \\ \quad \mathtt{trigger}\ (\mathsf{LWr}^{()}\ (\uparrow id, v)) \\ \{v_s'.\ \langle \mathsf{id} := v \rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ (\{[\%id]\} \cup L)\}^{\mathsf{src}} \end{array}$$

We are currently at stack frame "i", and we can extend the local domain and get a new local environment predicate.

Function calls in Vellvm

Function calls, stack-allocated resources



```
(* The denotation of an itree function is a coq function that takes
   a list of uvalues and returns the appropriate semantics. *)
Definition function_denotation : Type :=
  list uvalue → itree L0' uvalue.

Definition denote_function (df:definition dtyp (cfg dtyp)) : function_denotation :=
  λ (args : list uvalue) ⇒
    (* We match the arguments variables to the inputs *)
    bs ←
      (* Allow only full application of functions *)
      (if Nat.eqb (List.length (df_args df)) (List.length args) then
        ret (List.combine (df_args df) args)
       else raise ("Incorrect argument length for function")) ;;
    (* generate the corresponding writes to the local stack frame *)
    trigger MemPush ;;
    trigger (StackPush bs) ;;
    rv ← translate instr_to_L0' (denote_cfg (df_instrs df)) ;;
    trigger StackPop ;;
    trigger MemPop ;;
    ret rv.
```

Stack-allocated resources

Function calls, stack-allocated resources

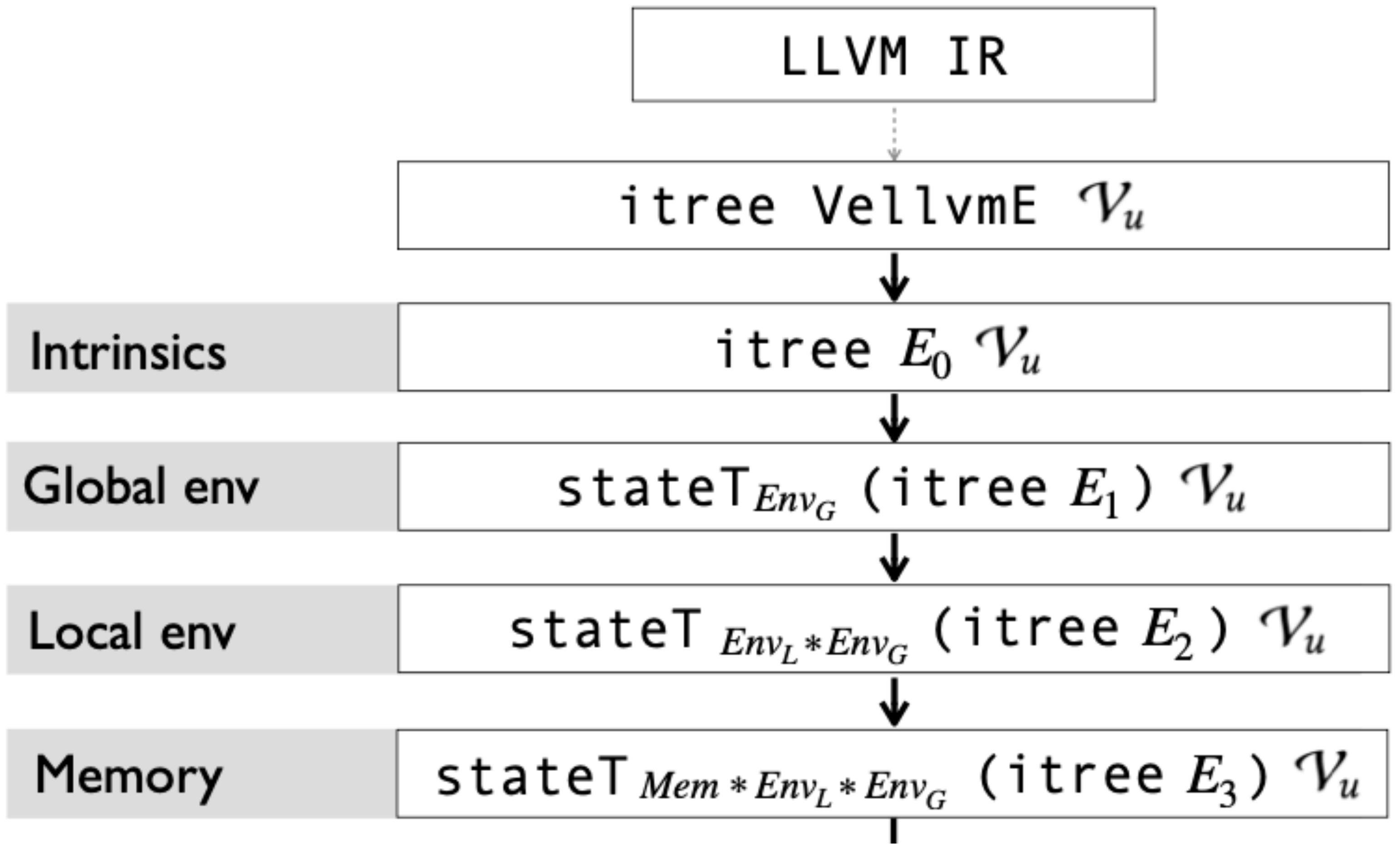
- Each function call allocates a new stack frame

$$
\begin{array}{l}
\textsc{SourcePushFrame} \\
\{\mathsf{Frame}^{\mathsf{src}}\ i\} \\
\quad \texttt{trigger}\ (\texttt{MPush}^{()})\ ;;\ \texttt{trigger}\ (\texttt{LPush}^{()}\,(\mathit{args})) \\
\{v_s'.\ \exists f.\mathsf{Frame}^{\mathsf{src}}\ (f :: i) * \mathsf{Alloca}_f^{\mathsf{src}}\ \emptyset * \mathsf{Local}_f^{\mathsf{src}}\ (\mathsf{dom}(\mathit{args})) * (\mathop{\ast}_{(id,v)\in \mathit{args}} \langle \mathsf{id} := v\rangle_f^{\mathsf{src}})\}^{\mathsf{src}}
\end{array}
$$

We are currently at stack frame "i", and if we push a new memory frame and local frame with arguments "args", we update the current frame index, and get a empty memory frame and local domain and ownership over arguments "args" pushed onto the stack.

Event rules

Atomic proof rules over events



Need to build atomic proof rules over the sum of events on VIR

Memory-relevant event rules

SourceAlloca

$$\{\mathsf{Alloca}_i^{\mathsf{src}}\ S * \mathsf{Frame}^{\mathsf{src}}\ i\}$$
$$\mathtt{trigger}\ (\mathtt{Alloca}^{\mathcal{V}}(\tau))$$
$$\{v'_s.\ \exists \ell_s. v_s = \ell_s * \ell_s \mapsto^{\mathsf{src}} \mathsf{new\_block}^{\tau} * \mathsf{Alloca}_i^{\mathsf{src}}\ \{z\} \cup S * \mathsf{Frame}^{\mathsf{src}}\ i\}^{\mathsf{src}}$$

SourceLoad

$$\{(v \in \tau) * \ell_s \mapsto^{\mathsf{src}} v\}$$
$$\mathtt{trigger}\ (\mathsf{Load}^{\mathcal{V}u}(\tau, \mathsf{addr}(\ell)))$$
$$\{v'_s.\ v'_s = v * \ell_s \mapsto^{\mathsf{src}} v\}^{\mathsf{src}}$$

SourceStore

$$\{(v \in \tau) * \ell_s \mapsto^{\mathsf{src}} v\}$$
$$\mathtt{trigger}\ (\mathsf{Store}^{()}(\mathsf{addr}(\ell), v'))$$
$$\{v'_s.\ \ell_s \mapsto^{\mathsf{src}} v'\}^{\mathsf{src}}$$

UB and Exception event rules

SimUB $e \preceq \mathtt{trigger}\ (\mathsf{UB}^{\emptyset})\{\Phi\}$

SimExc $e \preceq \mathtt{trigger}\ (\mathsf{Throw}^{\emptyset})\{\Phi\}$

"Undefined behavior subsumes all behavior"

and

"The simulation holds only if the source program does not go wrong"

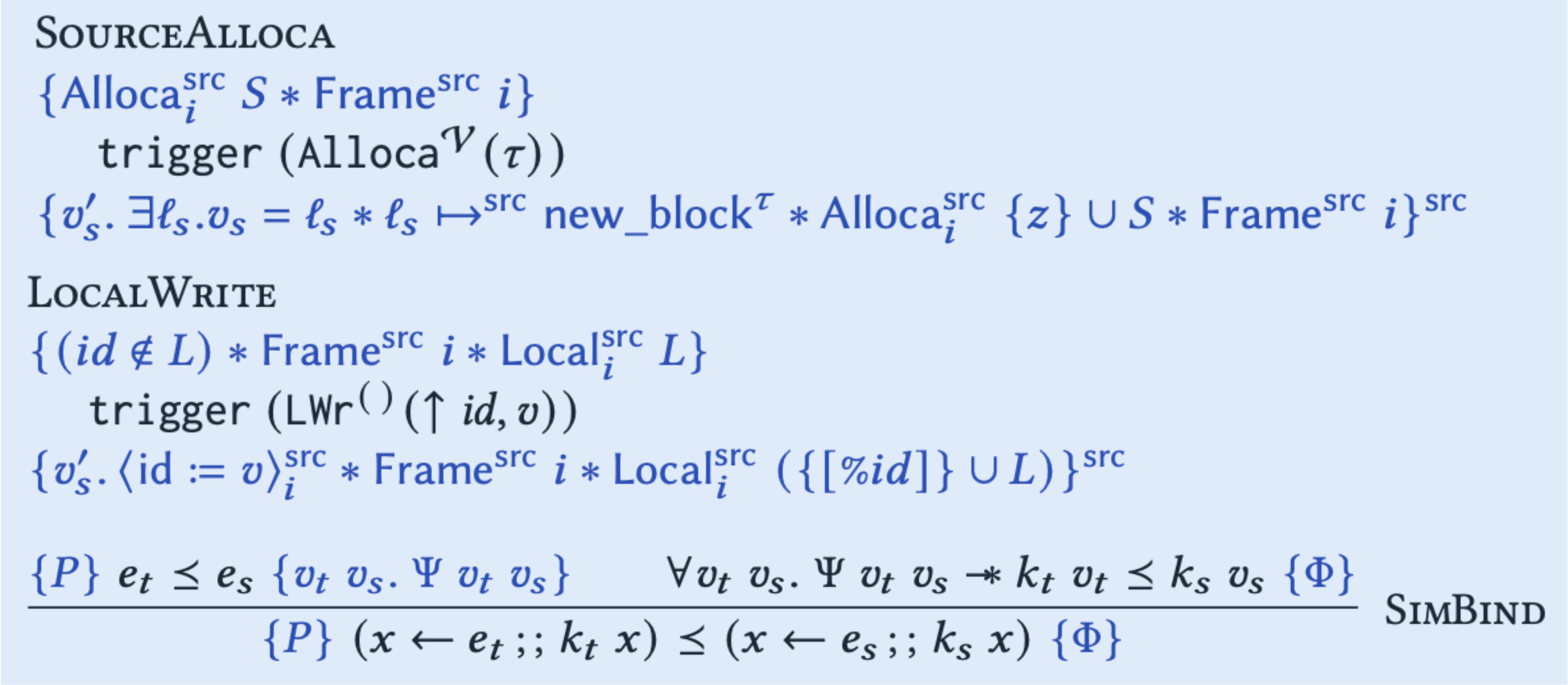
Instruction rules

Example: Alloca instruction

- Given atomic proof rules, it is straightforward to build rules over denotations on syntax

$$
\begin{aligned}
&[\![(id, \texttt{alloca}\ (\tau))]\!]_i = dv \leftarrow \\
&\quad \texttt{trigger}\ (\texttt{Alloca}^{\mathcal{V}}(\tau))\ ;; \\
&\quad \texttt{trigger}\ (\texttt{LWr}^{()}(\uparrow id, dv))
\end{aligned}
$$

Denotation of an alloca instruction



SourceAlloca

$\{\mathsf{Alloca}_i^{\mathsf{src}}\ S * \mathsf{Frame}^{\mathsf{src}}\ i\}$

$\quad \texttt{trigger}\ (\texttt{Alloca}^{\mathcal{V}}(\tau))$

$\{v_s'.\ \exists \ell_s. v_s = \ell_s * \ell_s \mapsto^{\mathsf{src}} \mathsf{new\_block}^{\tau} * \mathsf{Alloca}_i^{\mathsf{src}}\ \{z\} \cup S * \mathsf{Frame}^{\mathsf{src}}\ i\}^{\mathsf{src}}$

LocalWrite

$\{(id \notin L) * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ L\}$

$\quad \texttt{trigger}\ (\texttt{LWr}^{()}(\uparrow id, v))$

$\{v_s'.\ \langle \mathsf{id} := v\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ (\{[\%id]\} \cup L)\}^{\mathsf{src}}$

$$\frac{\{P\}\ e_t \preceq e_s\ \{v_t\ v_s.\ \Psi\ v_t\ v_s\} \qquad \forall v_t\ v_s.\ \Psi\ v_t\ v_s \mathrel{-\!\!*} k_t\ v_t \preceq k_s\ v_s\ \{\Phi\}}{\{P\}\ (x \leftarrow e_t\ ;;\ k_t\ x) \preceq (x \leftarrow e_s\ ;;\ k_s\ x)\ \{\Phi\}}\ \textsc{SimBind}$$

SourceInstrAlloca

$\{(x \notin L) * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Alloca}_i^{\mathsf{src}}\ S * \mathsf{Local}_i^{\mathsf{src}}\ L\}$

$\quad \%x^{\mathsf{Id}} = \mathsf{alloca}\ \tau$

$\{\exists \ell_s.\ \ell \mapsto^{\mathsf{src}} \mathsf{new\_block}^{\tau} * \langle \mathsf{x} := \mathsf{addr}(\ell)\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Alloca}_i^{\mathsf{src}}\ S * \mathsf{Local}_i^{\mathsf{src}}\ (\{[\%x]\} \cup L)\}^{\mathsf{src}}$

Roadmap

1. A taste of the stack-based ghost theory for LLVM IR ✅
2. Memory-relevant attributes in LLVM IR
3. Model : A relational weakest-precondition model for Interaction Trees in Iris
4. Adequacy

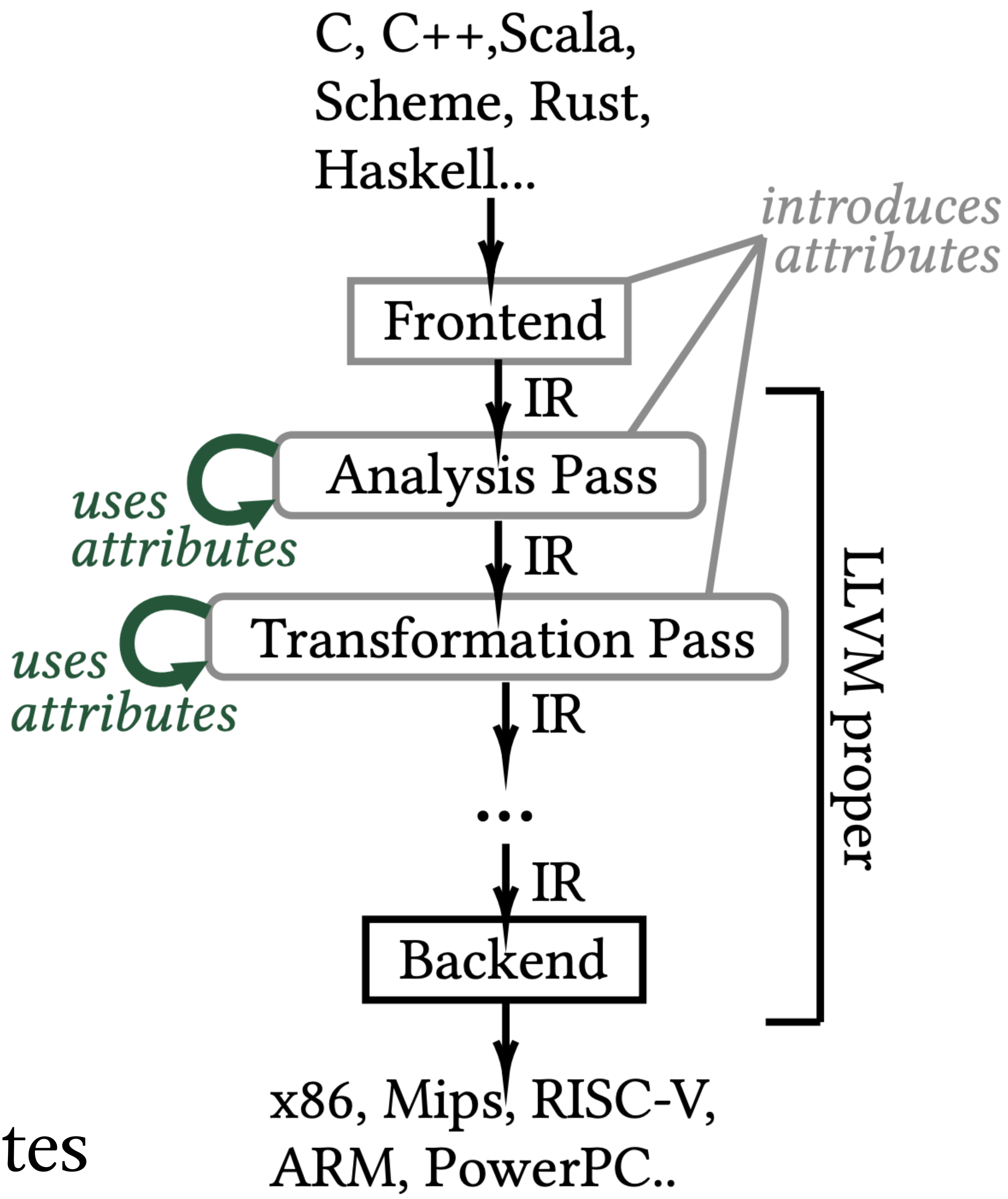
Memory attributes in LLVM IR

- LLVM optimization and analysis passes often use memory attributes, lightweight specifications about how a function may affect memory

```
define void @f(i32*) readonly argmemonly { ... }
```

" function f only reads from arguments passed on to the function "

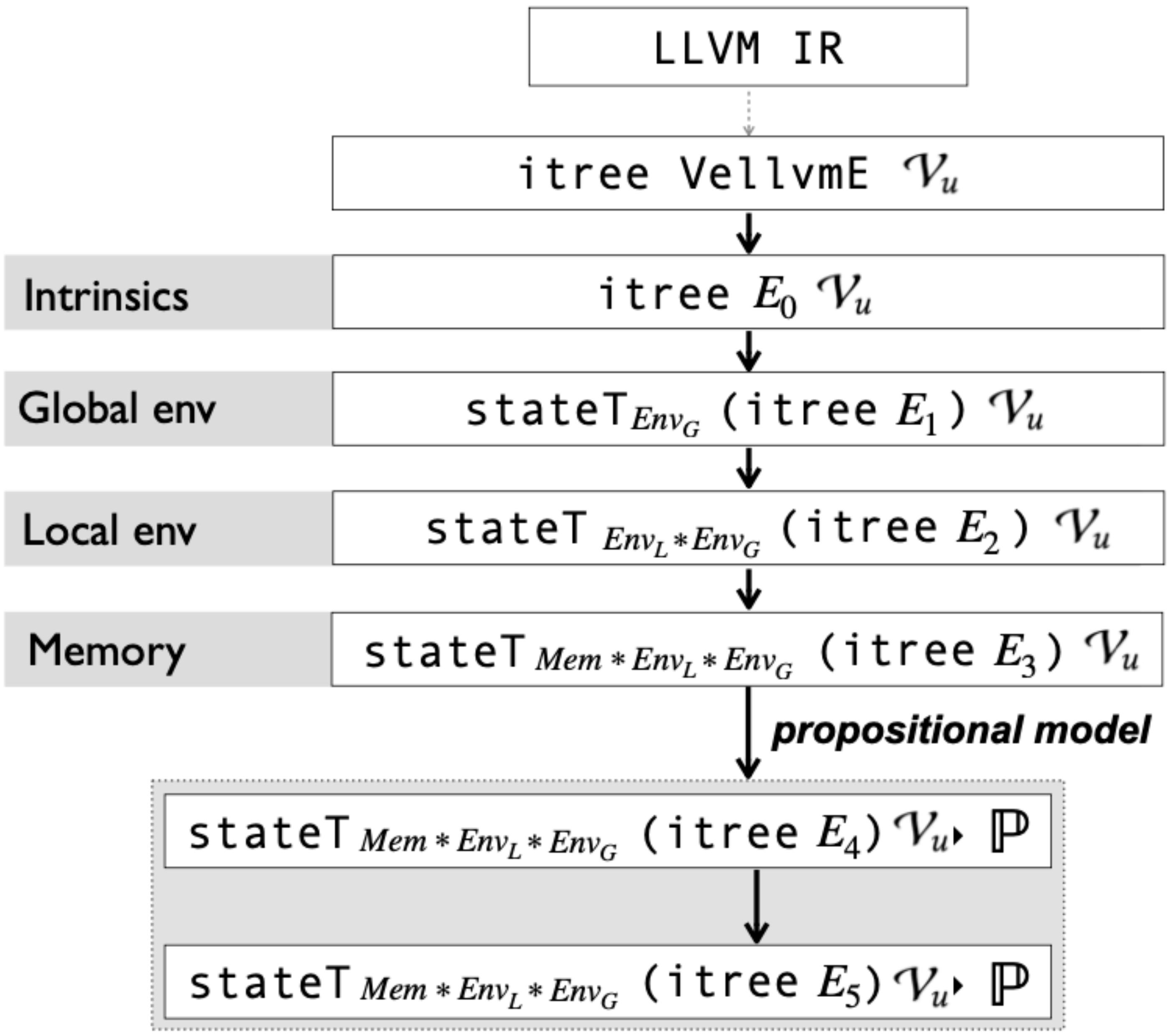
- Logical interpretation of memory attributes using permission-based ownership
- Can reason about reordering across calls and transformations that take advantage of memory attributes



External call semantics

Let's fix the naïve semantics!

- Pre-existing VIR semantics: external calls could not affect memory



LLVM IR

`itree VellvmE` $\mathcal{V}_u$

Intrinsics — `itree` E_0 $\mathcal{V}_u$

Global env — `stateT`$_{Env_G}$ `(itree` E_1`)` $\mathcal{V}_u$

Local env — `stateT`$_{Env_L * Env_G}$ `(itree` E_2`)` $\mathcal{V}_u$

Memory — `stateT`$_{Mem * Env_L * Env_G}$ `(itree` E_3`)` $\mathcal{V}_u$

propositional model

`stateT`$_{Mem * Env_L * Env_G}$ `(itree` E_4`)` $\mathcal{V}_{u}$ $\mathbb{P}$

`stateT`$_{Mem * Env_L * Env_G}$ `(itree` E_5`)` $\mathcal{V}_{u}$ $\mathbb{P}$

- Event transformer: transforms an event into a state-passing event

```
Variant stateEff (E : Type -> Type) : Type -> Type :=
| StateEff {X} : S * E X -> stateEff (S * X).
```

With this simple change, external calls are aware of memory

How do we reason about function calls?

- \`eutt\` is not enough: we cannot rely on syntactic trace equivalence for function calls
  - e.g. `call foo (%p1)` can be related to `call foo (%p2)` if `%p1` and `%p2` store related pointers

Simuliris [Gäher et al.]-style public resources

- Locations in public bijection (related pointers) $\ell_t \leftrightarrow_h \ell_s$

LocEscape

$$\frac{\{\ell_t \leftrightarrow_h \ell_s * P\}\ e_t \preceq e_s\ \{\Phi\}}{\{\ell_t \mapsto^{\mathsf{tgt}} v_t * \ell_s \mapsto^{\mathsf{src}} v_s * \mathcal{V}(v_t, v_s) * P\}\ e_t \preceq e_s\ \{\Phi\}}$$

How to reason about public resources

first approximation: Simuliris [Gäher et al.]-style public resources

- We can store and load from public resources if they haven't been checked out by others yet

Has this been checked out yet?

Simstore

$$\{\ell_t \leftrightarrow_h \ell_s * \mathcal{V}(v_t, v_s) * \text{checkout } C * (\ell_t, \ell_s) \notin C\}$$
$$\quad \texttt{trigger}\ (\texttt{Store}^{()}(\texttt{addr}(\ell_t), v_t)) \preceq \texttt{trigger}\ (\texttt{Store}^{()}(\texttt{addr}(\ell_s), v_s))$$
$$\{v_t, v_s. \mathcal{V}(v_t, v_s) * \text{checkout } C\}$$

SimLoad

$$\{\ell_t \leftrightarrow_h \ell_s * \text{checkout } C * ((\ell_t, \ell_s) \notin C \vee C(\ell_t, \ell_s) < 1)\}$$
$$\quad \texttt{trigger}\ (\texttt{Load}^{\mathcal{V}_u}(\tau, \texttt{addr}(\ell_t))) \preceq \texttt{trigger}\ (\texttt{Load}^{\mathcal{V}_u}(\tau, \texttt{addr}(\ell_s)))$$
$$\{v_t, v_s. \mathcal{V}(v_t, v_s) * \text{checkout } C\}$$

Function attribute specifications

Attribute specifications

SIMCALL

$$\{\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathsf{checkout}\ \emptyset * \overrightarrow{\mathcal{V}}(args_t, args_s)\}$$
$$\mathsf{call}\ \tau\ f(args_t) \ \preceq\ \mathsf{call}\ \tau\ f(args_s)$$
$$\{v_t, v_s. \mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathsf{checkout}\ \emptyset * \mathcal{V}(v_t, v_s)\}$$

Nothing's been checked out, so anyone can have full access to public resources!

- Based on the function attribute, the simulation checks whether the patron should have full access to the material (or a partial scan) (i.e. permission-based ownership), and makes sure that all resources have been safely returned.

READONLY-CALL

$$\{\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathsf{checkout}\ C *$$
$$* \overrightarrow{\mathcal{V}}(args_t, args_s) * (\forall(\ell_t, \ell_s) \in C. C(\ell_t, \ell_s) = q \land q < 1)\}$$
$$\mathsf{call}\ \tau\ f(args_t)\ \mathsf{readonly} \ \preceq\ \mathsf{call}\ \tau\ f(args_s)\ \mathsf{readonly}$$
$$\{v_t, v_s. \mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathcal{V}(v_t, v_s) * \mathsf{checkout}\ C\}$$

Let's check the access privileges ...

Roadmap

1. A taste of the stack-based ghost theory for LLVM IR ✅
2. Memory-relevant attributes in LLVM IR ✅
3. Model : A relational weakest-precondition model for Interaction Trees in Iris
4. Adequacy

Building an Iris framework for VIR

Typical recipe

(1) Ingredient: an abstract view on state (ghost theory) using separation logic resources

(2) Ingredient: a small-step semantics

Given a small-step semantics, a Hoare triple can be derived via the typical weakest precondition model* of Iris

(technically, a Banach guarded fixpoint)*

What we have (and need)

(1) Ingredient: A ghost theory for VIR resources

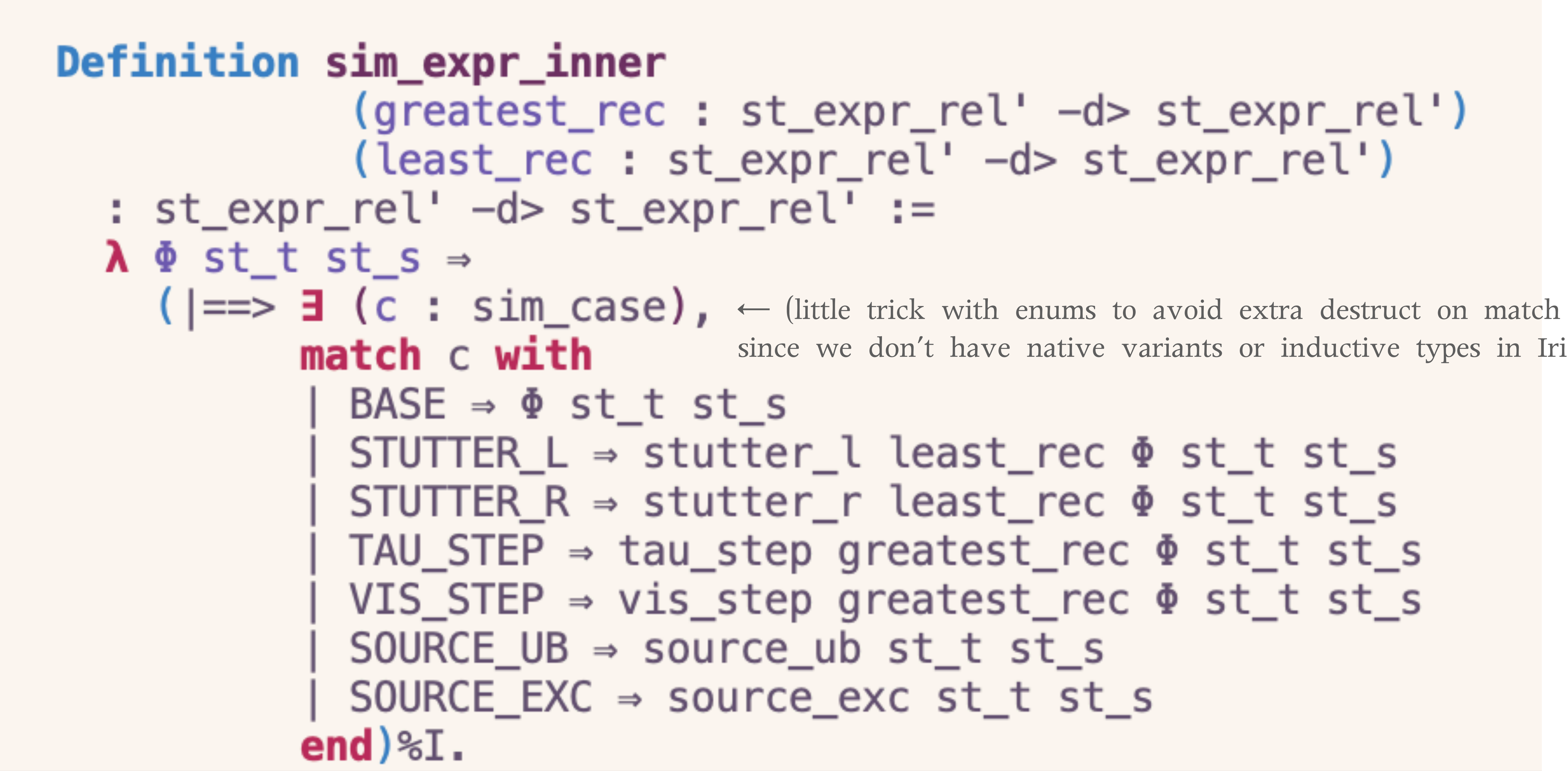
(2) Ingredient: ITree-based semantics

A new weakest precondition model* of Iris for stateful ITrees

(technically, a Knaster-Tarski mixed fixpoint)*

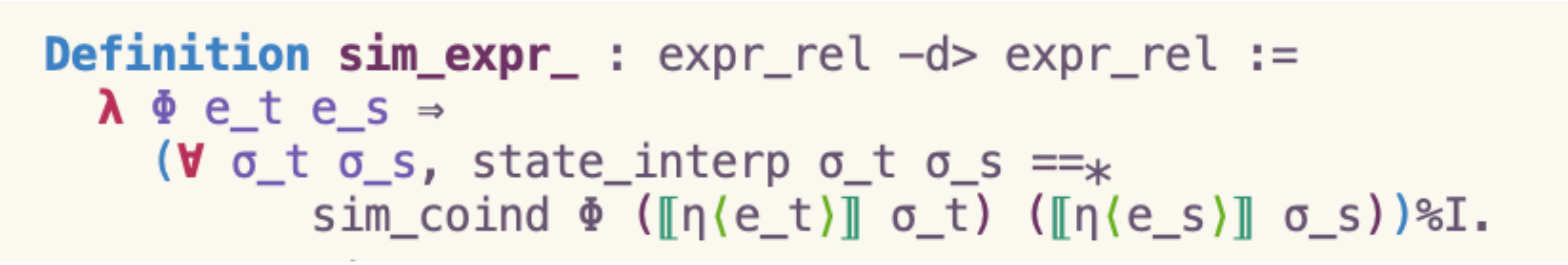
Weakest precondition

Behind the scenes...



```
Definition sim_expr_inner
            (greatest_rec : st_expr_rel' -d> st_expr_rel')
            (least_rec : st_expr_rel' -d> st_expr_rel')
  : st_expr_rel' -d> st_expr_rel' :=
  λ Φ st_t st_s ⇒
   (|==> ∃ (c : sim_case),
         match c with
         | BASE ⇒ Φ st_t st_s
         | STUTTER_L ⇒ stutter_l least_rec Φ st_t st_s
         | STUTTER_R ⇒ stutter_r least_rec Φ st_t st_s
         | TAU_STEP ⇒ tau_step greatest_rec Φ st_t st_s
         | VIS_STEP ⇒ vis_step greatest_rec Φ st_t st_s
         | SOURCE_UB ⇒ source_ub st_t st_s
         | SOURCE_EXC ⇒ source_exc st_t st_s
         end)%I.
```

← (little trick with enums to avoid extra destruct on match cases, since we don't have native variants or inductive types in Iris)

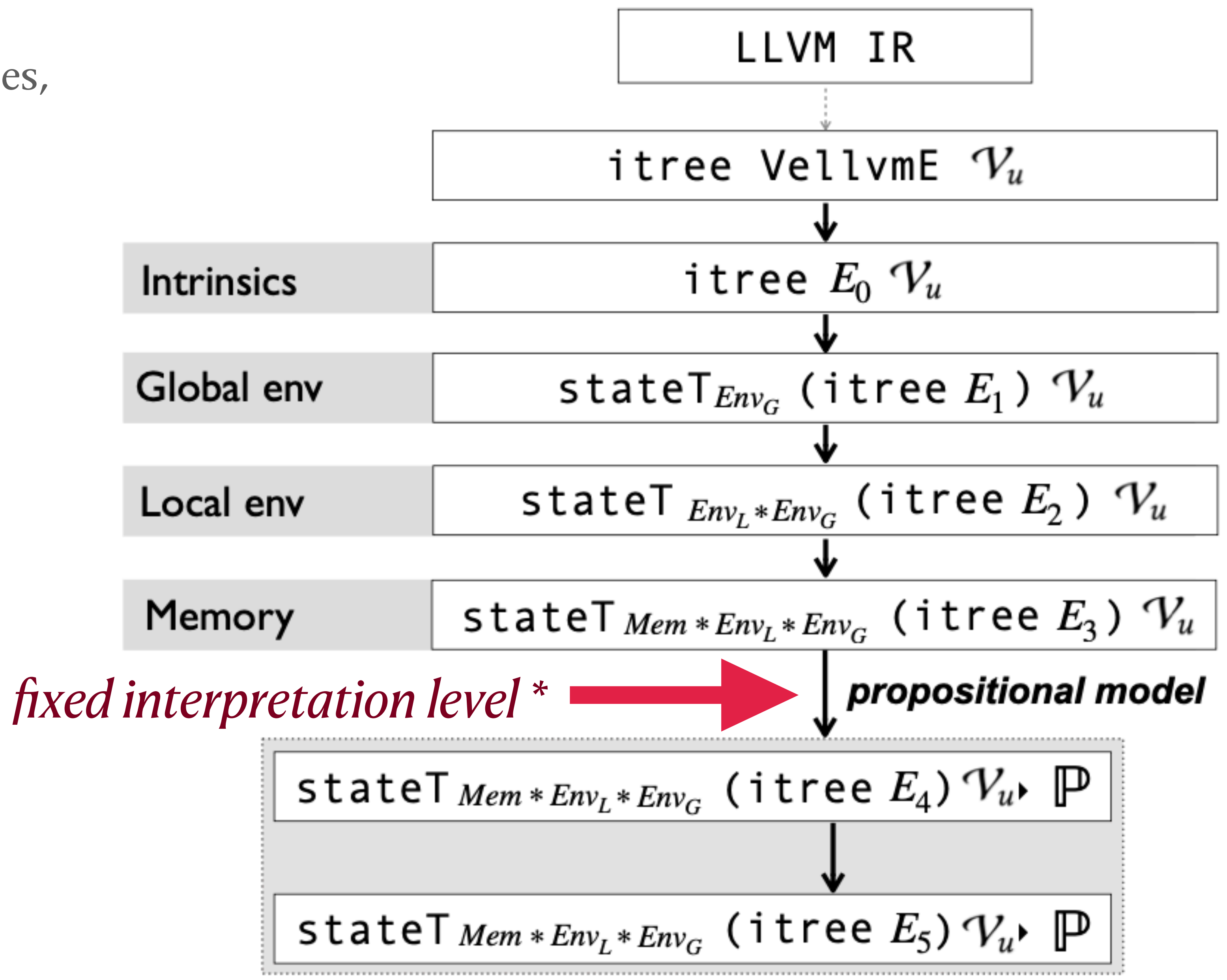


```
Definition sim_expr_ : expr_rel -d> expr_rel :=
  λ Φ e_t e_s ⇒
   (∀ σ_t σ_s, state_interp σ_t σ_s ==∗
        sim_coind Φ (⟦η⟨e_t⟩⟧ σ_t) (⟦η⟨e_s⟩⟧ σ_s))%I.
```

(sim__coind takes the mixed greatest-least fixpoint of sim__expr__inner)

Q. What is the stateful interpretation function ⟦ - ⟧ ?

A.



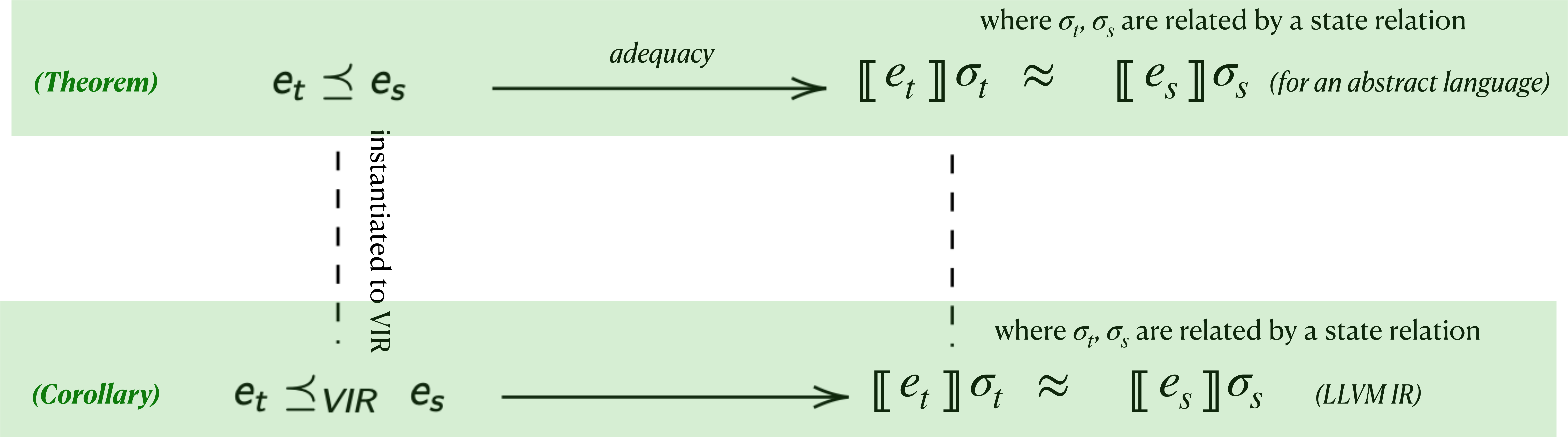
(with modified interpretation for calls)

Roadmap

1. A taste of the stack-based ghost theory for LLVM IR ✅
2. Memory-relevant attributes in LLVM IR ✅
3. Model : A relational weakest-precondition model for Interaction Trees in Iris ✅
4. Adequacy

Adequacy

For ITrees e_t and e_s without external calls,



where σ_t, σ_s are related by a state relation

(Theorem) $e_t \preceq e_s \xrightarrow{adequacy} [\![e_t]\!]\sigma_t \approx [\![e_s]\!]\sigma_s$ *(for an abstract language)*

instantiated to VIR

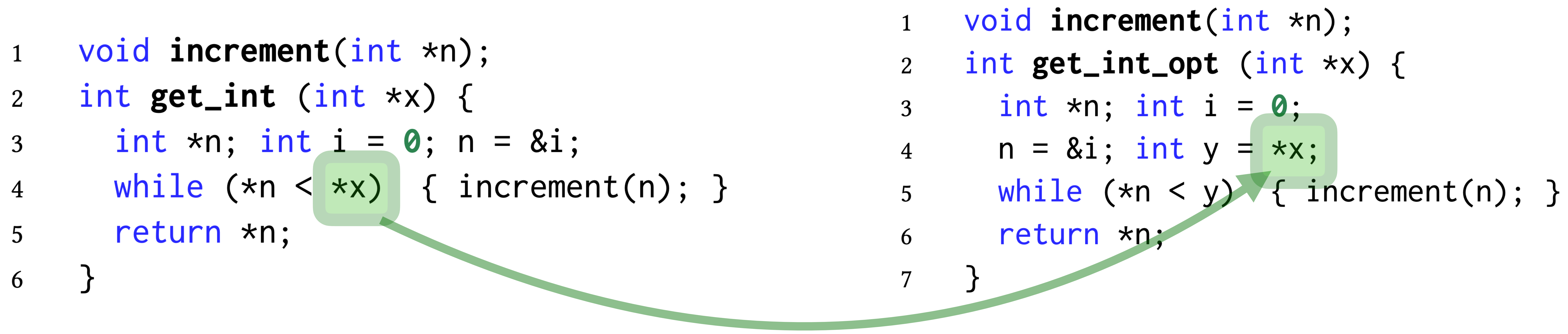
where σ_t, σ_s are related by a state relation

(Corollary) $e_t \preceq_{VIR} e_s \longrightarrow [\![e_t]\!]\sigma_t \approx [\![e_s]\!]\sigma_s$ *(LLVM IR)*

Roadmap

1. A taste of the stack-based ghost theory for LLVM IR ✅
2. Memory-relevant attributes in LLVM IR ✅
3. Model : A relational weakest-precondition model for Interaction Trees in Iris ✅
4. Adequacy ✅

Back to: Proving LICM



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

- Example: proof of simulation for a simple loop invariant code motion algorithm
- Benefits: can reorder memory-relevant instructions with function calls (could not be expressed before)
- Benefits: Hoare-style reasoning over loops; proof does not require explicit coinduction

Contributions

: results mechanized in the Coq Proof Assistant
Development: ~ 40k LOC (and ongoing...)

Velliris: A relational separation logic framework for LLVM IR

- A relational, coinductive weakest precondition model of Iris which supports a monadic semantics based on the Interaction Trees framework
- A relational separation logic and ghost theory for VIR resources
- Logical interpretation for memory-relevant attributes
- Examples: collection of simple examples and proof of simple loop invariant code motion algorithm
- Logical relation and contextual refinement (omitted)
- (Ongoing) case study: Verification of Mem2Reg algorithm

Thank you!