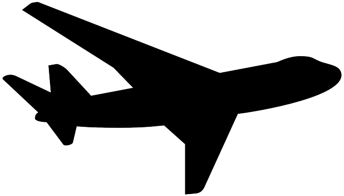
A Concurrency Model based on Monadic Interpreters: Executable semantics for a concurrent subset of LLVM IR

PhD defense

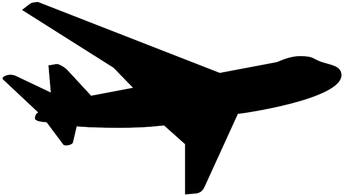
Nicolas Chappe (Inria Lyon, LIP)
Supervised by Ludovic Henrio (CNRS, LIP) and Yannick Zakowski (Inria Lyon, LIP)

November 22, 2024

Computer programs are everywhere, including in critical systems that have to be bug-free



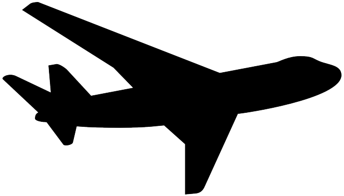
Computer programs are everywhere, including in critical systems that have to be bug-free



The code the machine understands is hard to understand for humans

```
        cmpl        $0, a(%rip)
        jl        .LBB0_2
# %bb.1:
        leaq        .L.str(%rip), %rdi
        movb        $0, %al
        callq        printf@PLT
.LBB0_2:
```

Computer programs are everywhere, including in critical systems that have to be bug-free



The code the machine understands is hard to understand for humans

```
        cmpl        $0, a(%rip)
        jl         .LBB0_2
# %bb.1:
        leaq        .L.str(%rip), %rdi
        movb        $0, %al
        callq        printf@PLT
.LBB0_2:
```

Verified compilation can reconcile these two facts

```
if (a >= 0)
  say_hello()
```

Human-readable code
written in a *programming language*

```
if (a >= 0)
  say_hello()
```

Human-readable code
written in a *programming language*

Compiler →

```
        cmpl        $0, a(%rip)
        jl        .LBB0_2
# %bb.1:
        leaq        .L.str(%rip), %rdi
        movb        $0, %al
        callq        printf@PLT
.LBB0_2:
```

Machine-readable code
in *assembly language*

- Is the compiler output correct?

$$\text{Compilation:}\qquad \begin{tikzcd} \overset{\text{source program}}{\mathtt{P}} \arrow[r] \arrow[d, bend left, red] & \overset{\text{compiled program}}{\mathscr{C}(\mathtt{P})} \\ \underset{\substack{\text{behaviors of}\\ \text{source program}}}{[\![\mathtt{P}]\!]} & \end{tikzcd}$$

❶ Formalize the behaviors of programs in the source language

- Is the compiler output correct?

source program | compiled program

Compilation:

$$\begin{tikzcd} \mathtt{P} \arrow[r] \arrow[d, bend left] & \mathscr{C}(\mathtt{P}) \arrow[d, bend right, red] \\ [\![\mathtt{P}]\!] & [\![\mathscr{C}(\mathtt{P})]\!] \end{tikzcd}$$

behaviors of source program | behaviors of compiled program

1. Formalize the behaviors of programs in the source language
2. Formalize the behaviors of programs in the target language

- Is the compiler output correct?

$$
\begin{array}{lccc}
 & \text{source program} & & \text{compiled program} \\
\text{Compilation:} & \mathtt{P} & \longrightarrow & \mathscr{C}(\mathtt{P}) \\
 & \downarrow & & \downarrow \\
\text{Verified compilation:} & [\![\mathtt{P}]\!] & \stackrel{?}{\supseteq} & [\![\mathscr{C}(\mathtt{P})]\!] \\
 & \text{behaviors of} & & \text{behaviors of} \\
 & \text{source program} & & \text{compiled program}
\end{array}
$$

1. Formalize the behaviors of programs in the source language
2. Formalize the behaviors of programs in the target language
3. Check that the behaviors of the compiled program are also behaviors of the source program

- Is the compiler output correct?

source program | compiled program

Compilation: $\mathtt{P} \longrightarrow \mathscr{C}(\mathtt{P})$

$$\begin{tikzcd} \mathtt{P} \arrow[r] \arrow[d, bend left] & \mathscr{C}(\mathtt{P}) \arrow[d, bend right] \\ [\![\mathtt{P}]\!] \arrow[r, phantom, "\overset{?}{\supseteq}"] & [\![\mathscr{C}(\mathtt{P})]\!] \end{tikzcd}$$

Verified compilation: $[\![\mathtt{P}]\!] \overset{?}{\supseteq} [\![\mathscr{C}(\mathtt{P})]\!]$

behaviors of source program | behaviors of compiled program

1. Formalize the behaviors of programs in the source language
2. Formalize the behaviors of programs in the target language
3. Check that the behaviors of the compiled program are also behaviors of the source program

- Notable example: CompCert in Coq/Rocq, compiles C to optimized machine code

```
if (a >= 0)
  say_hello()
```

Human-readable code
that satisfies some safety properties

$\xrightarrow{\text{Verified compiler}}$

```
        cmpl      $0, a(%rip)
        jl        .LBB0_2
# %bb.1:
        leaq      .L.str(%rip), %rdi
        movb      $0, %al
        callq     printf@PLT
.LBB0_2:
```

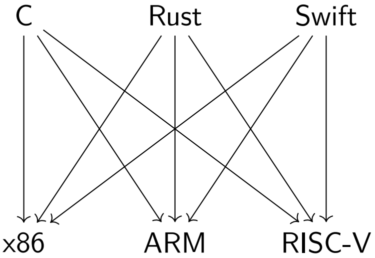
Machine-readable code
that satisfies the same safety properties

Many programming languages C Rust Swift

Many programming languages	C	Rust	Swift
Many kinds of machine (*architectures*)	x86	ARM	RISC-V

Many programming languages

Many kinds of machine
(*architectures*)



This is getting complex!

LLVM IR

Many programming languages — C, Rust, Swift

↓

Intermediate representation — LLVM IR

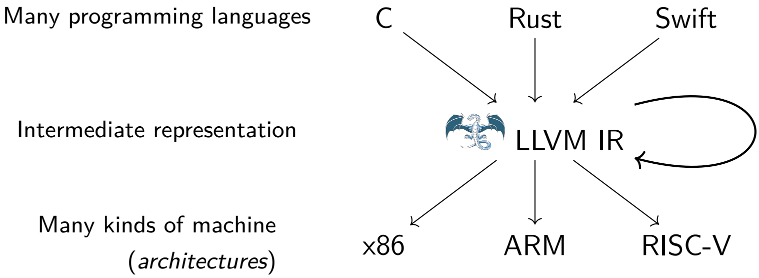
↓

Many kinds of machine (*architectures*) — x86, ARM, RISC-V

- LLVM IR: intermediate language that factors out many compilers[1]

[1] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: CGO '04. USA: IEEE Computer Society, 2004.

LLVM IR

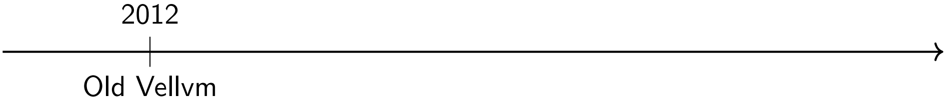


- LLVM IR: intermediate language that factors out many compilers[1]
- Many program transformations and optimizations defined on LLVM IR
- Interesting starting point for a verified compilation infrastructure

[1] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: CGO '04. USA: IEEE Computer Society, 2004.

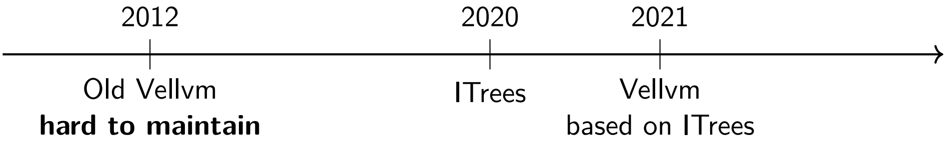
Related work: Vellvm

Vellvm's goal: *efficiently model the semantics of LLVM IR in Rocq*



Related work: Vellvm

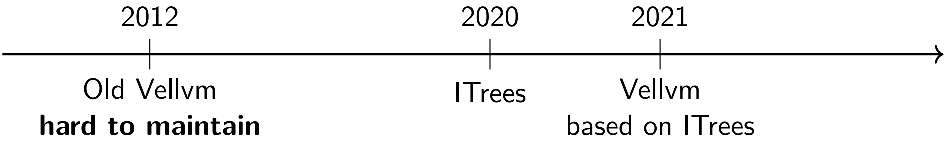
Vellvm's goal: *efficiently model the semantics of LLVM IR in Rocq*[2]



[2] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. "Modular, Compositional, and Executable Formal Semantics for LLVM IR". In: *Proc. ACM Program. Lang.* 5.ICFP (Aug. 2021). DOI: 10.1145/3473572.

Related work: Vellvm

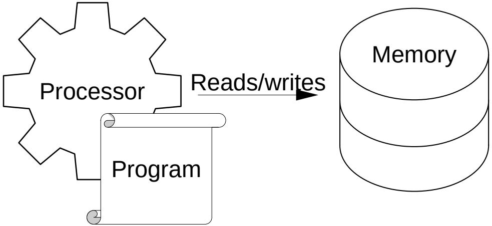
Vellvm's goal: *efficiently model the semantics of LLVM IR in Rocq*[2]



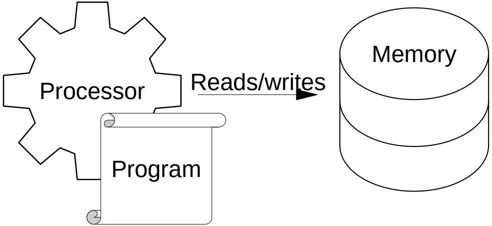
Limitation: *no support for concurrency*

[2] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. "Modular, Compositional, and Executable Formal Semantics for LLVM IR". In: *Proc. ACM Program. Lang.* 5.ICFP (Aug. 2021). DOI: 10.1145/3473572.

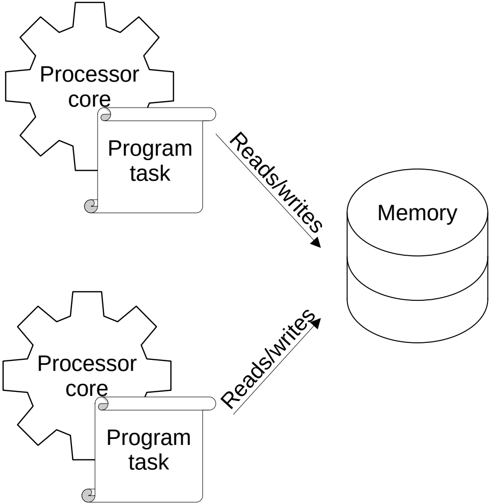
Standard program: series of instructions for the processor



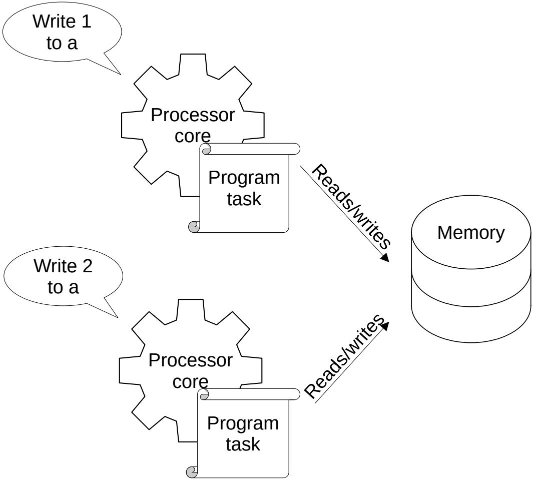
Standard program: series of instructions for the processor



Concurrent program: contains several tasks (or threads)



What happens when two tasks access the same memory cell?



- Many other subtleties, cf. *weak memory models*

Contributions

Vellvm's goal: *efficiently model the semantics of LLVM IR in Rocq*

2012	2020	2021
Old Vellvm **hard to maintain**	ITrees	Vellvm based on ITrees

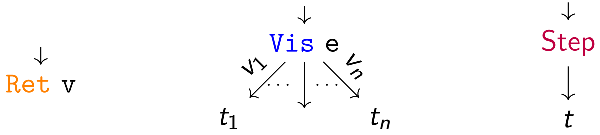
Our goal: *efficiently model the semantics of **concurrent** LLVM IR in Rocq*

2023	2024
CTrees	Concurrent LLVM semantics based on CTrees

Context: Interaction Trees

Interaction Trees (ITrees)[3]

- Tree model for representing programs
- Modular, reusable semantics
- Executable semantics
- Mechanized as a Rocq (formerly Coq) library



Return a value | Interact with the environment | Perform an internal computation

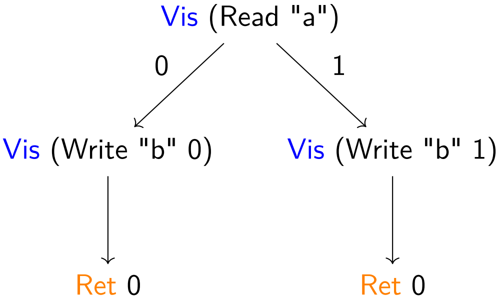
[3] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. "Interaction trees: representing recursive and impure programs in Coq". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371119.

ITrees: Visible events

- Each Vis node represents an effect of the program
- Branches of a Vis node represent the possible answers from the environment
- Ret nodes are leaves returning a value

LLVM IR example with memory access effects:

```
%x = load @a
store %x, @b
ret 0
```



ITrees are *coinductive*, they can be infinitely deep.

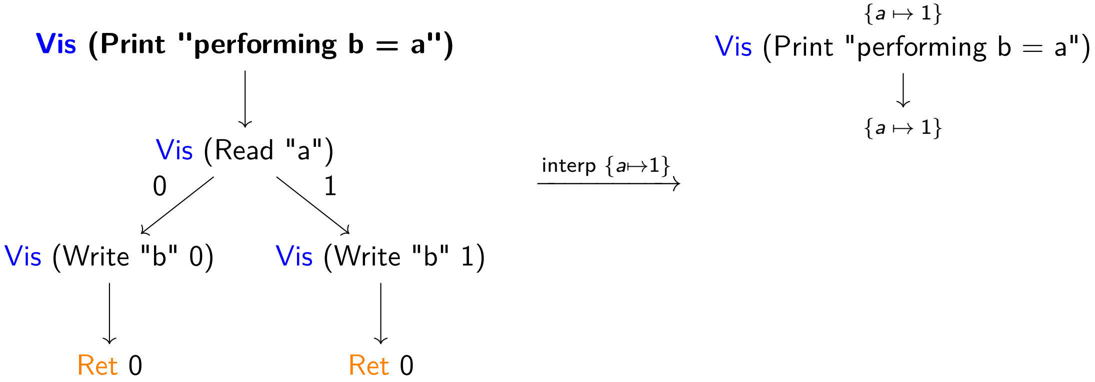
```
while true do print
```

Vis Print
↓
Step
↓
Vis Print
↓
Step
⋮

- Unary Step nodes mark internal events such as loop iterations

We can make the semantics appear by providing a stateful interpreter for Vis nodes.

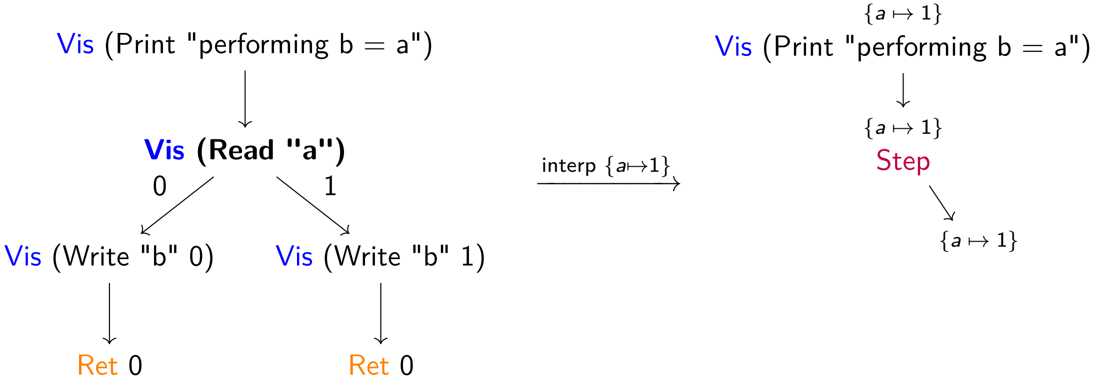
- Write events update a memory state
- Read events return the corresponding value from memory



ITrees: Interpretation

We can make the semantics appear by providing a stateful interpreter for Vis nodes.

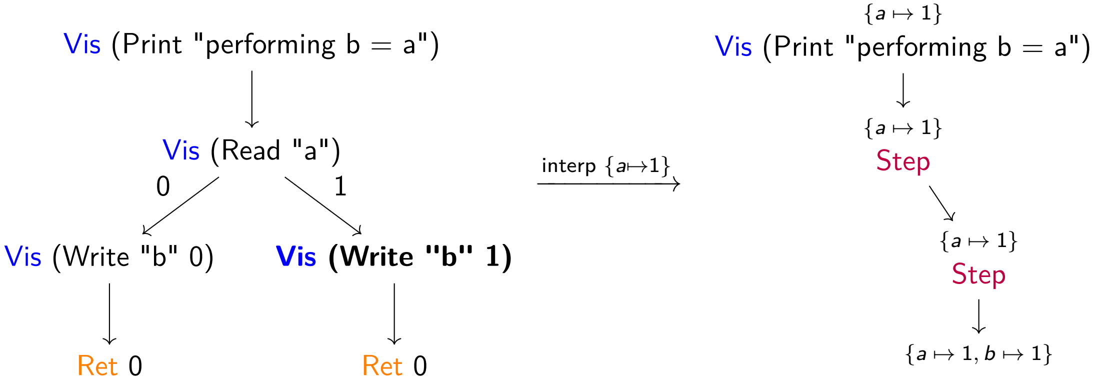
- Write events update a memory state
- Read events return the corresponding value from memory



ITrees: Interpretation

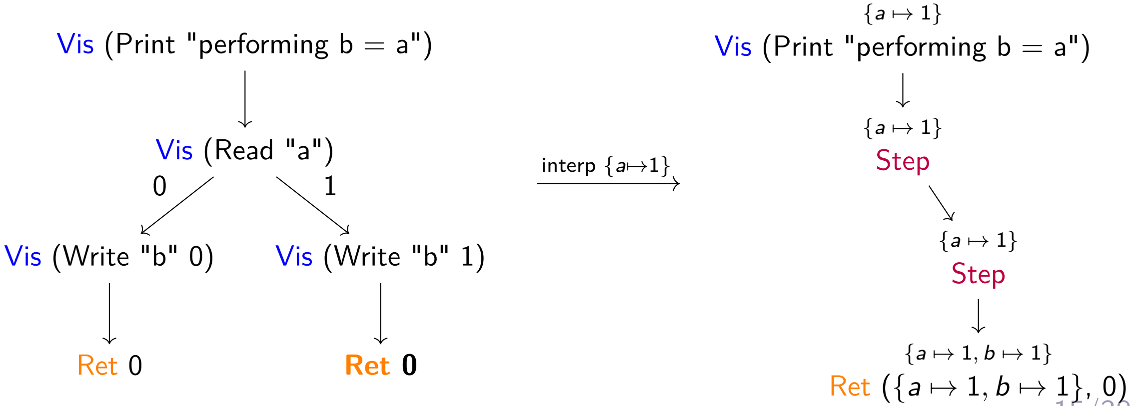
We can make the semantics appear by providing a stateful interpreter for Vis nodes.

- Write events update a memory state
- Read events return the corresponding value from memory

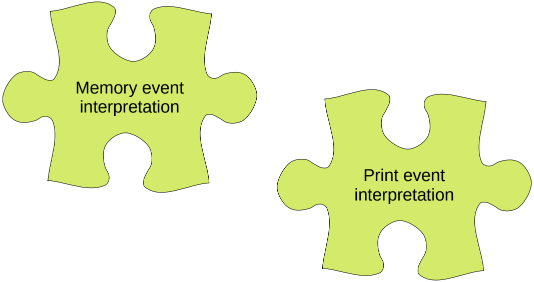


We can make the semantics appear by providing a stateful interpreter for Vis nodes.

- Write events update a memory state
- Read events return the corresponding value from memory



- General methodology: successively refine each kind of event node.



- In the end, we can execute the tree

$$\text{initial state} \mapsto \begin{array}{c} \textsf{Step} \\ \downarrow \\ \cdots \\ \downarrow \\ \textsf{Step} \\ \downarrow \\ \textsf{Ret}\ \text{final state} \end{array}$$

Vellvm applies this approach to LLVM, with 6 layers of interpretation

Choice Trees

Nondeterminism

Consider a `Rand` event that generates a random number. How to interpret it?

$$
\begin{tikzcd}
& \mathsf{Vis}\ \mathsf{Rand} \arrow[dl, "0"'] \arrow[dr, "1"] & \\
{} & & {}
\end{tikzcd}
$$

- No proper way to interpret it with ITrees
- Not interpreting it is semantically wrong

$$
\begin{tikzcd}
& \mathsf{Vis}\ \mathsf{Rand} \arrow[dl, "0"'] \arrow[dr, "1"] & & & & \mathsf{Vis}\ \mathsf{Rand} \arrow[dl, "0"'] \arrow[dr, "1"] & \\
\mathsf{Ret}\ 0 & & \mathsf{Ret}\ 1 & \not\approx & \mathsf{Ret}\ 1 & & \mathsf{Ret}\ 0
\end{tikzcd}
$$

Consider a `Rand` event that generates a random number. How to interpret it?

$$\begin{tikzcd} & \text{Br Rand} \arrow[dl, "0"'] \arrow[dr, "1"] & \\ {} & & {} \end{tikzcd}$$

- No proper way to interpret it with ITrees
- Not interpreting it is semantically wrong

$$\begin{tikzcd} & \text{Vis Rand} \arrow[dl, "0"'] \arrow[dr, "1"] & \\ \text{Ret } 0 & & \text{Ret } 1 \end{tikzcd} \not\approx \begin{tikzcd} & \text{Vis Rand} \arrow[dl, "0"'] \arrow[dr, "1"] & \\ \text{Ret } 1 & & \text{Ret } 0 \end{tikzcd}$$

- Special Br nodes represent nondeterministic choices, we will give them appropriate semantics later

Choice Trees (CTrees)

- Infinite (coinductive) tree model for representing programs
- Rocq library that takes inspiration from ITrees, with nondeterminism support
- Collaboration between LIP and UPenn

- Infinite (coinductive) tree model for representing programs
- Rocq library that takes inspiration from ITrees, with nondeterminism support
- Collaboration between LIP and UPenn

We can give nondeterministic semantics to Br nodes by defining a fitting notion of equivalence of CTrees

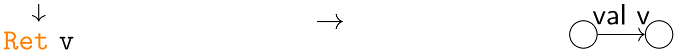
Building a labeled transition system (LTS) from a CTree

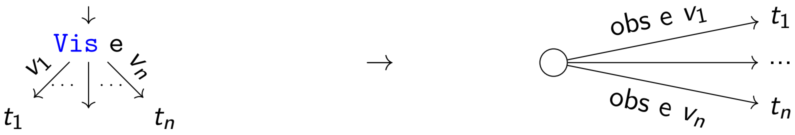
An LTS consists of a set of states and a set of labeled transitions.

$$\begin{array}{c}\downarrow\\ \texttt{Ret v}\end{array} \quad \longrightarrow \quad \bigcirc \xrightarrow{\text{val v}} \bigcirc$$

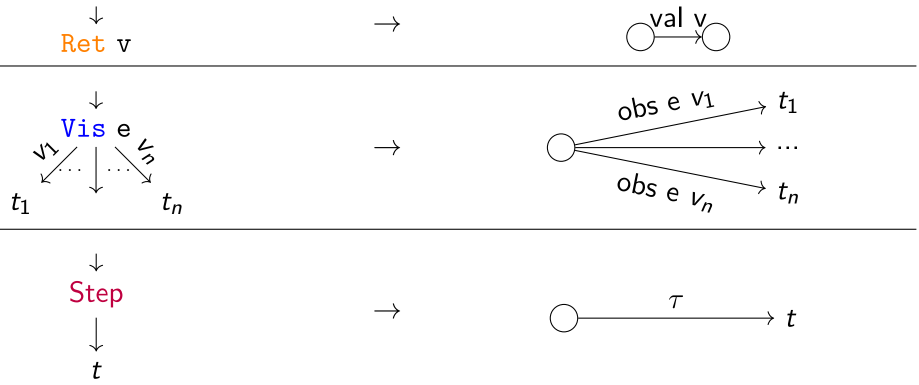
Building a labeled transition system (LTS) from a CTree

An LTS consists of a set of states and a set of labeled transitions.

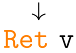




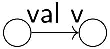
An LTS consists of a set of states and a set of labeled transitions.

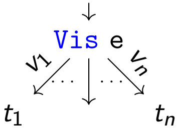


An LTS consists of a set of states and a set of labeled transitions.

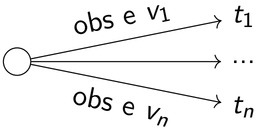


→

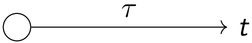


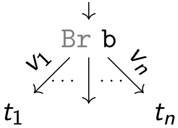


→



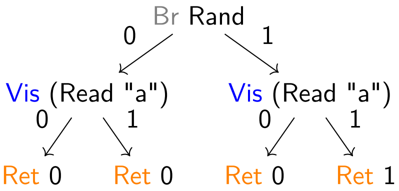
→



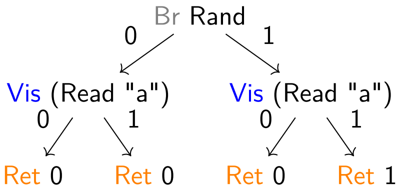


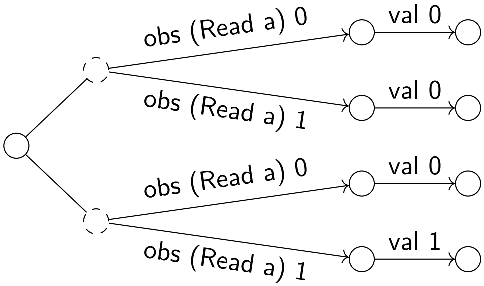
→

- Collapsed in the LTS
- Does not generate a transition

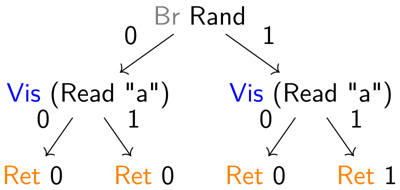

Br Rand
0
1
Vis (Read "a")
Vis (Read "a")
0
1
0
1
Ret 0
Ret 0
Ret 0
Ret 1

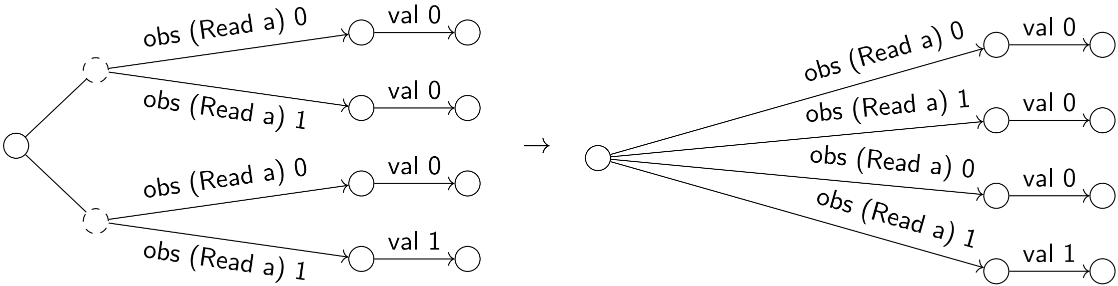
CTrees and the underlying LTS: example





CTrees and the underlying LTS: example





Legend

—— $\forall$

- - - $\exists$

$$
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}"'] \arrow[d, "a"'] & u \\
t' &
\end{tikzcd}
$$

Definition of a strong bisimulation $t \; \mathcal{R} \; u$, with two half-games

Legend

— $\forall$

- - - $\exists$

$$\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}"'] \arrow[d, "a"'] & u \arrow[d, dashed, "a"] \\
t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u'
\end{tikzcd}$$

Definition of a strong bisimulation $t \mathrel{\mathcal{R}} u$, with two half-games

CTree equivalence: Strong bisimulation on the LTS

Legend

—— $\forall$

- - - $\exists$

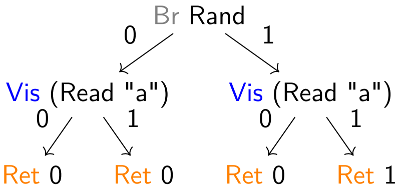
$$
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}"'] \arrow[d, "a"'] & u \arrow[d, dashed, "a"] \\
t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u'
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}"'] \arrow[d, dashed, "a"'] & u \arrow[d, "a"] \\
t' \arrow[r, dash, dashed, "\mathcal{R}"'] & u'
\end{tikzcd}
$$

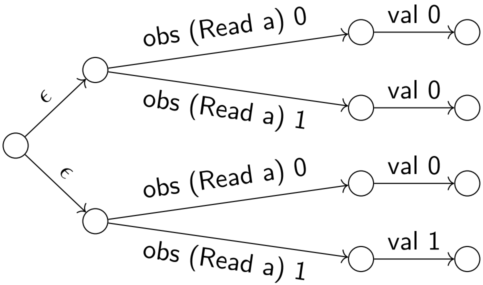
Definition of a strong bisimulation $t \ \mathcal{R} \ u$, with two half-games

- Br nodes verify the algebraic laws of non-determinism
- Equational proof principles for coinductive proofs
- Many up-to principles, using the coinduction library from Damien Pous

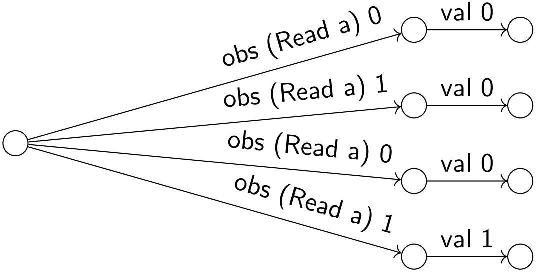
- We cannot reason about collapsed Br nodes, which complicated some of my proofs.
- We can build an LTS in which they appear.

- We cannot reason about collapsed Br nodes, which complicated some of my proofs.
- We can build an LTS in which they appear.





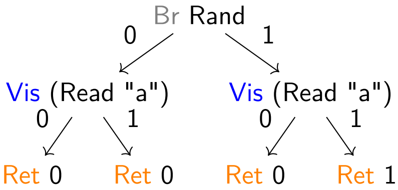
Br as ϵ transitions: *explicit LTS*

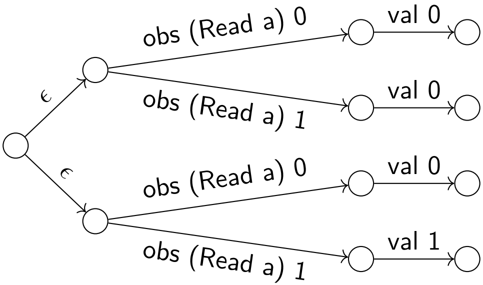


Br nodes collapsed in the original LTS

CTrees and the explicit LTS

- We cannot reason about collapsed Br nodes, which complicated some of my proofs.
- We can build an LTS in which they appear.





Br as ϵ transitions: *explicit LTS*

How do we define bisimulation on this alternative explicit LTS?
It should relate the same CTrees as the original definition.

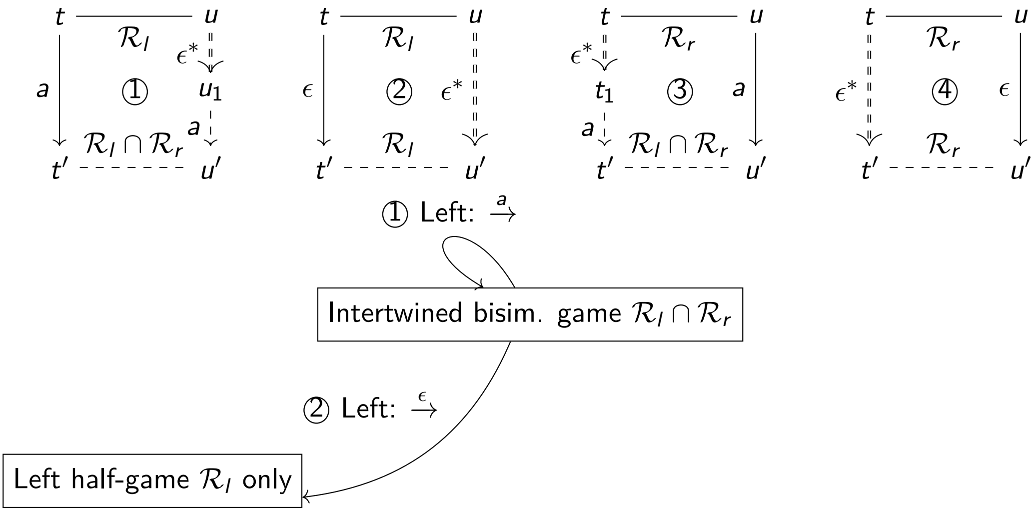
$$\begin{tikzcd}
t \arrow[rr, dash, "\mathcal{R}_l"'] \arrow[dd, "a"'] & & u \arrow[d, Rightarrow, dashed, "\epsilon^*"'] \\
& 1 & u_1 \arrow[d, dashed, "a"'] \\
t' \arrow[rr, dash, dashed, "\mathcal{R}_l \cap \mathcal{R}_r"] & & u'
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[rr, dash, "\mathcal{R}_l"'] \arrow[dd, "\epsilon"'] & & u \arrow[dd, Rightarrow, dashed, "\epsilon^*"'] \\
& 2 & \\
t' \arrow[rr, dash, dashed, "\mathcal{R}_l"] & & u'
\end{tikzcd}$$

$$\begin{tikzcd} t \arrow[r, dash, "\mathcal{R}_l"'] \arrow[dd, "a"'] & u \arrow[d, Rightarrow, dashed, "\epsilon^*"'] \\ \arrow[r, phantom, "①"] & u_1 \arrow[d, dashed, "a"'] \\ t' \arrow[r, dash, dashed, "\mathcal{R}_l \cap \mathcal{R}_r"] & u' \end{tikzcd} \qquad \begin{tikzcd} t \arrow[r, dash, "\mathcal{R}_l"'] \arrow[d, "\epsilon"'] \arrow[dr, phantom, "②"] & u \arrow[d, Rightarrow, dashed, "\epsilon^*"'] \\ t' \arrow[r, dash, dashed, "\mathcal{R}_l"] & u' \end{tikzcd} \qquad \begin{tikzcd} t \arrow[r, dash, "\mathcal{R}_r"'] \arrow[d, Rightarrow, dashed, "\epsilon^*"'] & u \arrow[dd, "a"'] \\ t_1 \arrow[d, dashed, "a"'] \arrow[r, phantom, "③"] & \\ t' \arrow[r, dash, dashed, "\mathcal{R}_l \cap \mathcal{R}_r"] & u' \end{tikzcd} \qquad \begin{tikzcd} t \arrow[r, dash, "\mathcal{R}_r"'] \arrow[d, Rightarrow, dashed, "\epsilon^*"'] \arrow[dr, phantom, "④"] & u \arrow[d, "\epsilon"'] \\ t' \arrow[r, dash, dashed, "\mathcal{R}_r"] & u' \end{tikzcd}$$

$$\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}_l"'] \arrow[dd, "a"'] \arrow[ddr, phantom, "1"] & u \arrow[d, Rightarrow, "\epsilon^*"'] \\
 & u_1 \arrow[d, dashed, "a"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}_l \cap \mathcal{R}_r"] & u'
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}_l"'] \arrow[d, "\epsilon"'] \arrow[dr, phantom, "2"] & u \arrow[d, Rightarrow, "\epsilon^*"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}_l"] & u'
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}_r"'] \arrow[d, Rightarrow, "\epsilon^*"'] \arrow[ddr, phantom, "3"] & u \arrow[dd, "a"] \\
t_1 \arrow[d, dashed, "a"'] & \\
t' \arrow[r, dash, dashed, "\mathcal{R}_l \cap \mathcal{R}_r"] & u'
\end{tikzcd}
\qquad
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}_r"'] \arrow[d, Rightarrow, "\epsilon^*"'] \arrow[dr, phantom, "4"] & u \arrow[d, "\epsilon"] \\
t' \arrow[r, dash, dashed, "\mathcal{R}_r"] & u'
\end{tikzcd}$$

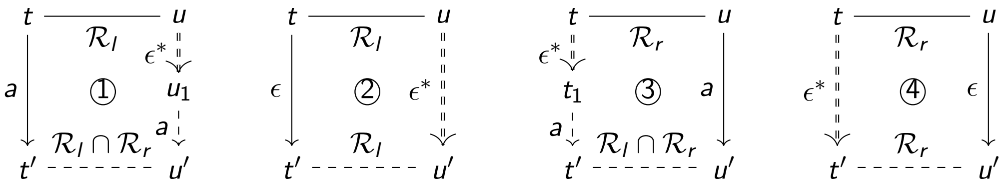
1 Left: $\xrightarrow{a}$

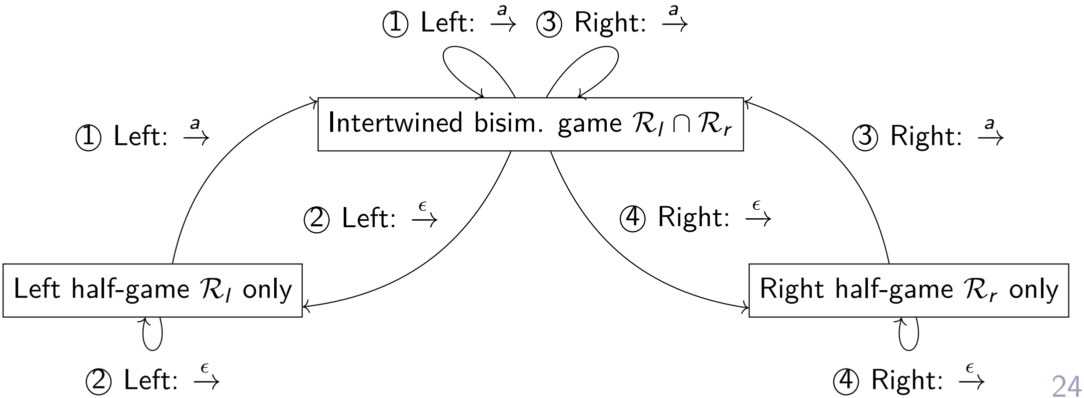
$\boxed{\text{Intertwined bisim. game } \mathcal{R}_l \cap \mathcal{R}_r}$


t $\mathcal{R}_l$ u ϵ^* u_1 a ① a $\mathcal{R}_l \cap \mathcal{R}_r$ t' u'
t $\mathcal{R}_l$ u ϵ ② ϵ^* $\mathcal{R}_l$ t' u'
t $\mathcal{R}_r$ u ϵ^* t_1 ③ a a $\mathcal{R}_l \cap \mathcal{R}_r$ t' u'
t $\mathcal{R}_r$ u ϵ^* ④ ϵ $\mathcal{R}_r$ t' u'
① Left: $\xrightarrow{a}$
Intertwined bisim. game $\mathcal{R}_l \cap \mathcal{R}_r$
② Left: $\xrightarrow{\epsilon}$
Left half-game $\mathcal{R}_l$ only

$$\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}_l"] \arrow[dd, "a"'] & u \arrow[d, Rightarrow, dashed, "\epsilon^*"'] \\
\arrow[r, phantom, "1"] & u_1 \arrow[d, dashed, "a"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}_l \cap \mathcal{R}_r"] & u'
\end{tikzcd}
\quad
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}_l"] \arrow[d, "\epsilon"'] \arrow[dr, phantom, "2"] & u \arrow[d, Rightarrow, dashed, "\epsilon^*"'] \\
t' \arrow[r, dash, dashed, "\mathcal{R}_l"] & u'
\end{tikzcd}
\quad
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}_r"] \arrow[d, Rightarrow, dashed, "\epsilon^*"'] & u \arrow[dd, "a"] \\
t_1 \arrow[d, dashed, "a"'] \arrow[r, phantom, "3"] & \\
t' \arrow[r, dash, dashed, "\mathcal{R}_l \cap \mathcal{R}_r"] & u'
\end{tikzcd}
\quad
\begin{tikzcd}
t \arrow[r, dash, "\mathcal{R}_r"] \arrow[d, Rightarrow, dashed, "\epsilon^*"'] \arrow[dr, phantom, "4"] & u \arrow[d, "\epsilon"] \\
t' \arrow[r, dash, dashed, "\mathcal{R}_r"] & u'
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{\text{Intertwined bisim. game } \mathcal{R}_l \cap \mathcal{R}_r} \arrow[loop above, "\text{1 Left: } \xrightarrow{a}"] \arrow[d, bend left, "\text{2 Left: } \xrightarrow{\epsilon}"] \\
\boxed{\text{Left half-game } \mathcal{R}_l \text{ only}} \arrow[u, bend left, "\text{1 Left: } \xrightarrow{a}"] \arrow[loop below, "\text{2 Left: } \xrightarrow{\epsilon}"]
\end{tikzcd}$$


t
u
$\mathcal{R}_l$
ϵ^*
u_1
a
①
a
$\mathcal{R}_l \cap \mathcal{R}_r$
t′
u′
t
u
$\mathcal{R}_l$
ϵ
②
ϵ^*
$\mathcal{R}_l$
t′
u′
t
u
ϵ^*
$\mathcal{R}_r$
t_1
③
a
a
$\mathcal{R}_l \cap \mathcal{R}_r$
t′
u′
t
u
$\mathcal{R}_r$
ϵ^*
④
ϵ
$\mathcal{R}_r$
t′
u′


① Left: $\xrightarrow{a}$ ③ Right: $\xrightarrow{a}$
Intertwined bisim. game $\mathcal{R}_l \cap \mathcal{R}_r$
① Left: $\xrightarrow{a}$
③ Right: $\xrightarrow{a}$
② Left: $\xrightarrow{\epsilon}$
④ Right: $\xrightarrow{\epsilon}$
Left half-game $\mathcal{R}_l$ only
Right half-game $\mathcal{R}_r$ only
② Left: $\xrightarrow{\epsilon}$
④ Right: $\xrightarrow{\epsilon}$

We define intertwined bisimulation using two *mutually coinductive* relations $\mathcal{R}_l$ and $\mathcal{R}_r$.

- Proved equivalent to strong bisimulation on the original LTS
- Makes some proofs significantly easier, especially around the interpretation combinator
- Equational theory similar to the original definition

- CTrees capture a wide class of nondeterministic LTSs
- Effects of a program can be handled in a modular way with interpretation
- CTrees are executable
- Paper at POPL'23[4]

Other personal contributions in the thesis

- Notions of refinement: Strong similarity, complete similarity
- Meta-theoretical results on interpretation and other combinators
- ...

[4] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. "Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq". In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571254.

Concurrent μVellvm

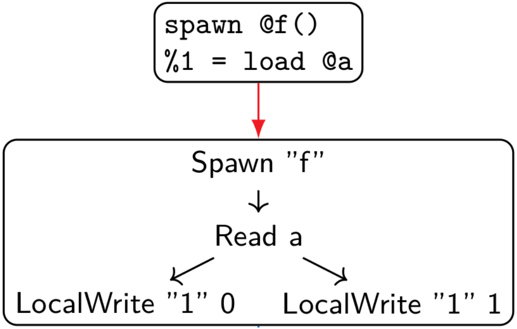
```
define @main() {
  store atomic 0, @x monotonic
  store atomic 0, @y monotonic
  call @thrd_create(@f)
  %1 = atomicrmw xchg @x, 1 monotonic
  %2 = atomicrmw xchg @y, 1 release
  ret 0
}

define @f() {
  %y1 = atomicrmw xchg @y, 2 acquire
  %x1 = load atomic @x monotonic
  ret 0
}
```

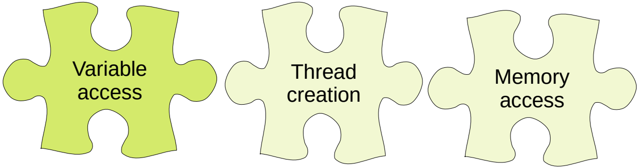
Syntax:

- Restricted subset of LLVM IR
- Thread management (`thrd_create` and `thrd_join`)
- Concurrent memory accesses

- We represent LLVM IR functions as CTrees with `(VarE + ThreadE + MemE)` events
- Reusable interpretation passes give semantics to these events

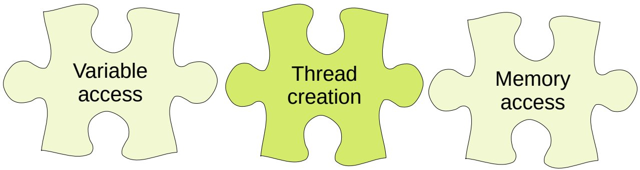


- We represent LLVM IR functions as CTrees with (**VarE** + ThreadE + MemE) events
- Reusable interpretation passes give semantics to these events



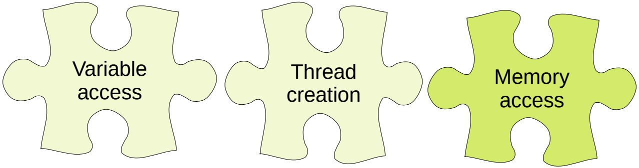

```
Variant VarE : Type → Type :=
| LocalWrite (id: ident) (v: value) : VarE unit
| LocalRead (id: ident) : VarE value
| GlobalRead (id: ident) : VarE value
```

- We represent LLVM IR functions as CTrees with `(VarE + ThreadE + MemE)` events
- Reusable interpretation passes give semantics to these events

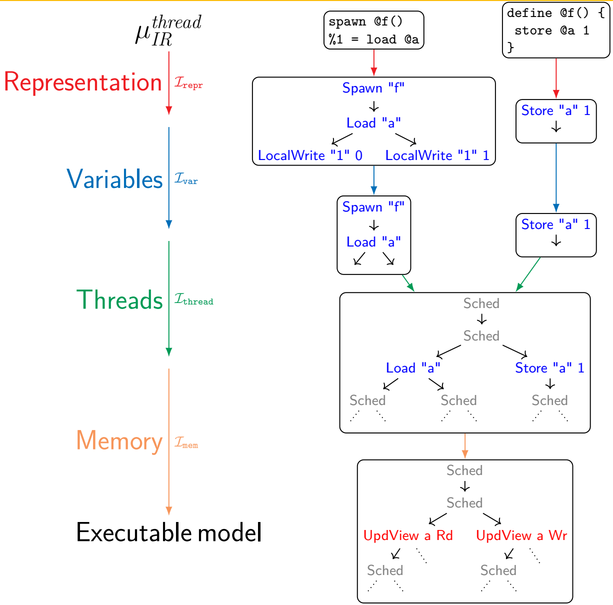



```
Variant ThreadE : Type → Type :=
| Spawn (f: fid) (arg: value) : ThreadE thread_id
```

- We represent LLVM IR functions as CTrees with (`VarE + ThreadE + MemE`) events
- Reusable interpretation passes give semantics to these events



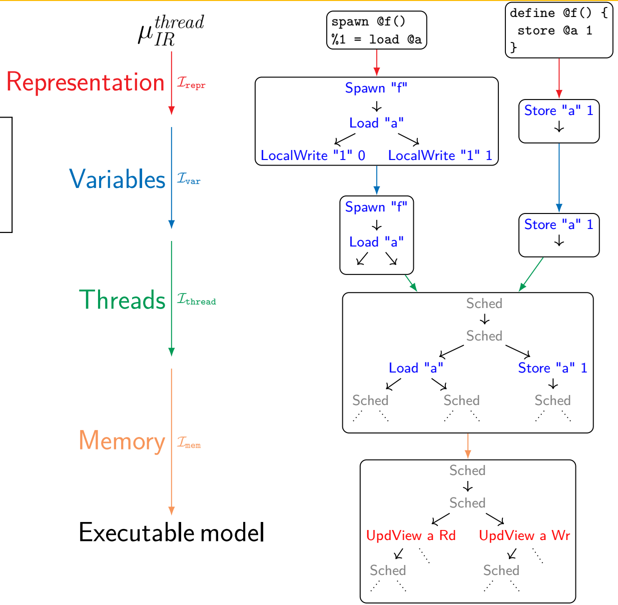

```
Variant MemE : Type → Type :=
| Load (o: ordering) (k: addr) : MemE value
| Store (o: ordering) (k: addr) (v: value) : MemE unit
| RMW (o: ordering) (k: addr) (f: value → value) : MemE value
| Fence (o: ordering) : MemE unit
| Alloc (sz: nat) : MemE addr
```

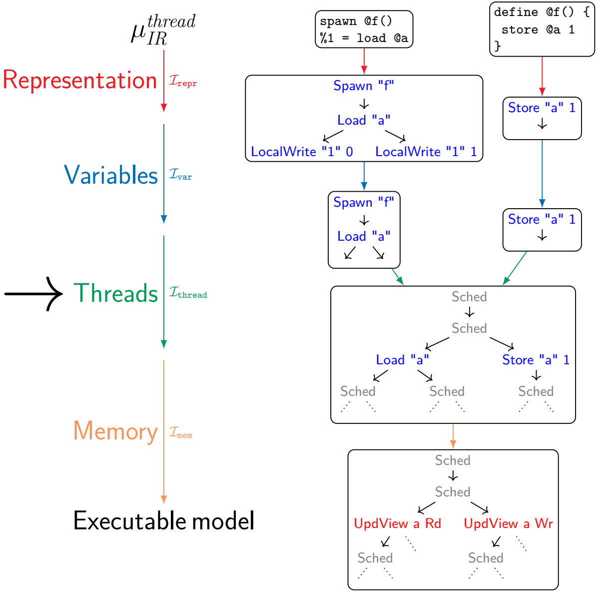


μ_{IR}^{thread}
Representation
$\mathcal{I}_{repr}$
Variables
$\mathcal{I}_{var}$
Threads
$\mathcal{I}_{thread}$
Memory
$\mathcal{I}_{mem}$
Executable model
spawn @f()
%1 = load @a
define @f() {
store @a 1
}
Spawn "f"
Load "a"
LocalWrite "1" 0
LocalWrite "1" 1
Store "a" 1
Spawn "f"
Load "a"
Store "a" 1
Sched
Sched
Load "a"
Store "a" 1
Sched
Sched
Sched
Sched
Sched
UpdView a Rd
UpdView a Wr
Sched
Sched

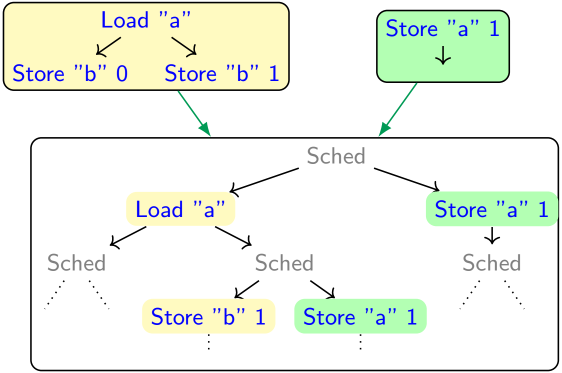
Interpretation stack

With a proof of transport of equivalences across the interpretation passes

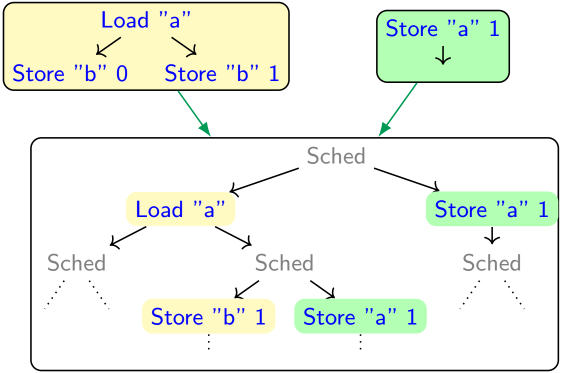



μ_{IR}^{thread}
Representation
$\mathcal{I}_{repr}$
Variables
$\mathcal{I}_{var}$
Threads
$\mathcal{I}_{thread}$
Memory
$\mathcal{I}_{mem}$
Executable model
spawn @f()
%1 = load @a
define @f() {
store @a 1
}
Spawn "f"
Load "a"
LocalWrite "1" 0
LocalWrite "1" 1
Store "a" 1
Sched
Load "a"
Store "a" 1
UpdView a Rd
UpdView a Wr

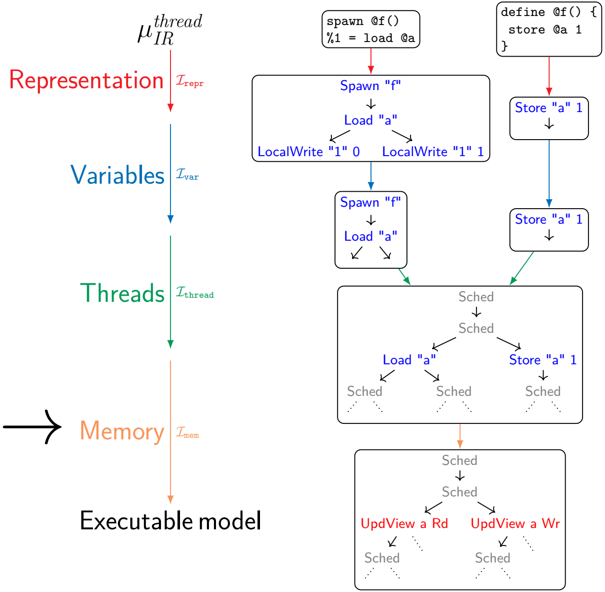
- To model the parallel composition of *n* threads, we interleave their CTrees, which gives a single CTree
- A `Br` `Sched` node chooses which thread to execute next

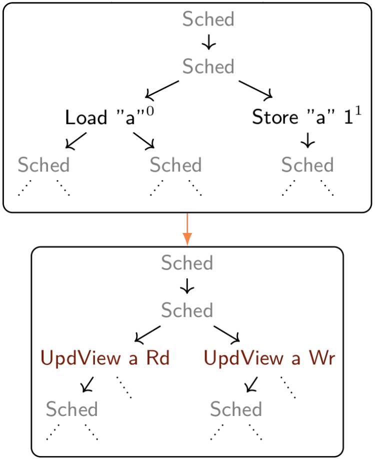


- To model the parallel composition of n threads, we interleave their CTrees, which gives a single CTree
- A `Br Sched` node chooses which thread to execute next
- Thread creation adds one CTree to the interleaving
- Based on a co-recursive combinator `interleave fns tasks`



Interpreting memory events



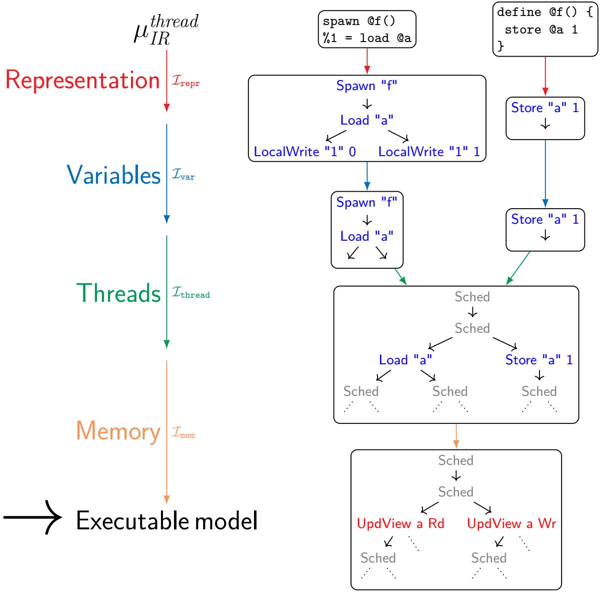


CTrees are modular! We support SC, TSO and a subset of Promising[a], with simulation results.

Promising-based model

- A very complete operational concurrent memory model
- We support read, write, read-modify-write and fence operations
- We support most levels of atomicity
- Monotonic accesses partly left to future work

[a] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. "A promising semantics for relaxed-memory concurrency". In: *POPL'17*. ACM, 2017. DOI: 10.1145/3009837.3009850.



- We can still execute CTrees, but we have to schedule the remaining branching nodes
- Round-robin, random...
- Collecting interpreter written in OCaml

Conclusion

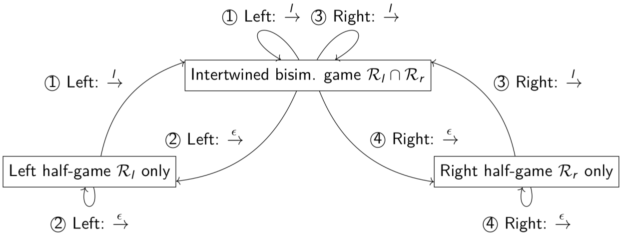
Contributions

- Choice Trees: modular and executable semantics for nondeterministic programs (POPL'23 paper)
- A novel notion of equivalence: intertwined bisimulation (paper in submission)
- Modular concurrency semantics, applied to LLVM (paper accepted at CPP'25)

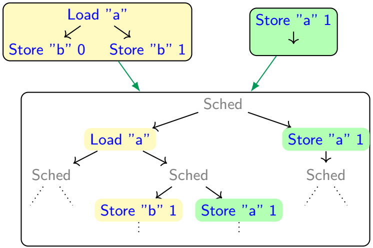
Future work

- Meta-theoretical results on the interpretation stack
- Integration into Vellvm
- A verified compilation chain

Choice Trees



Intertwined bisimulation



LLVM IR concurrency

Thanks for your attention!