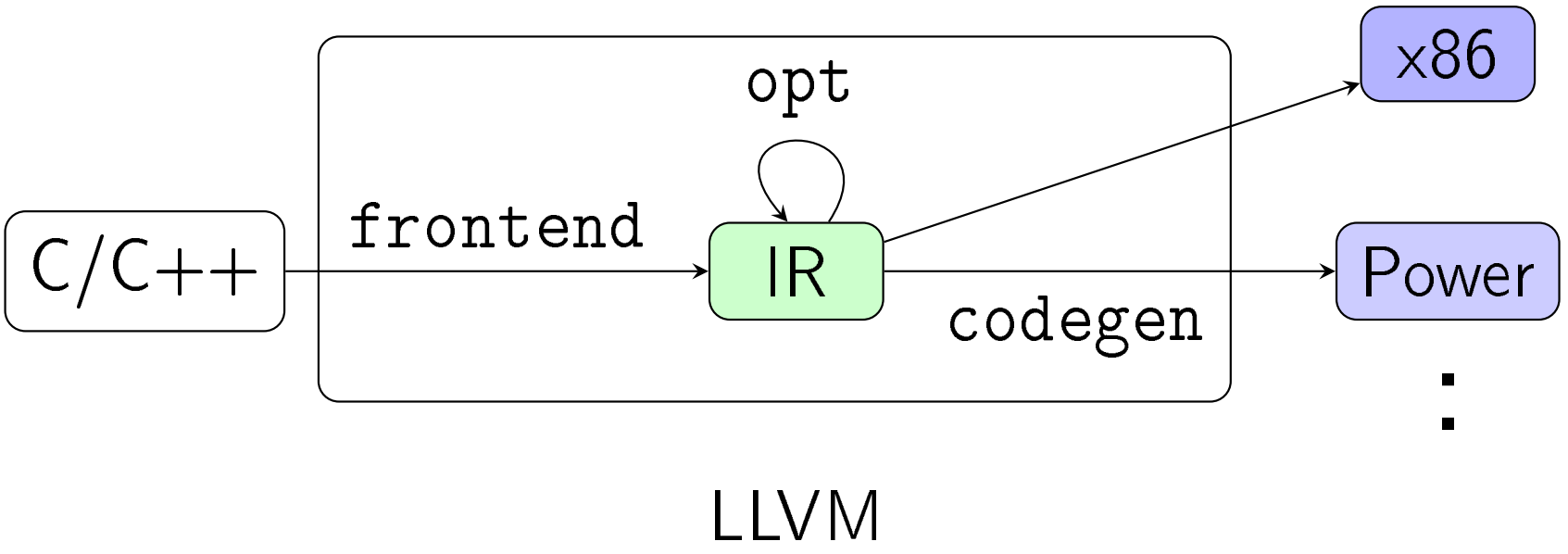
Formalizing the Concurrency Semantics of an LLVM Fragment

Soham Chakraborty, Viktor Vafeiadis

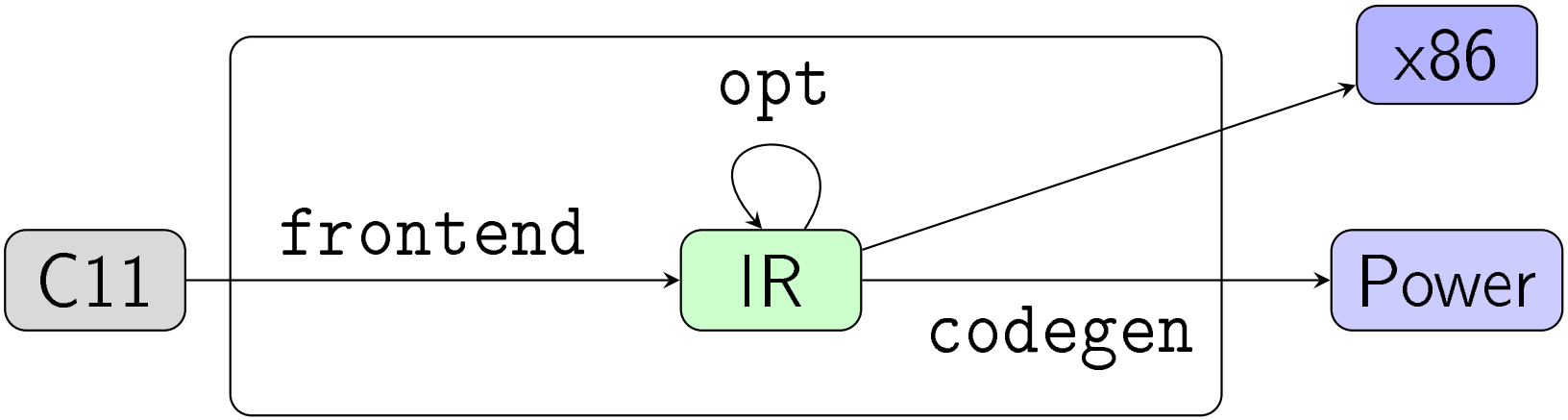
Max Planck Institute for Software Systems (MPI-SWS)

CGO 2017

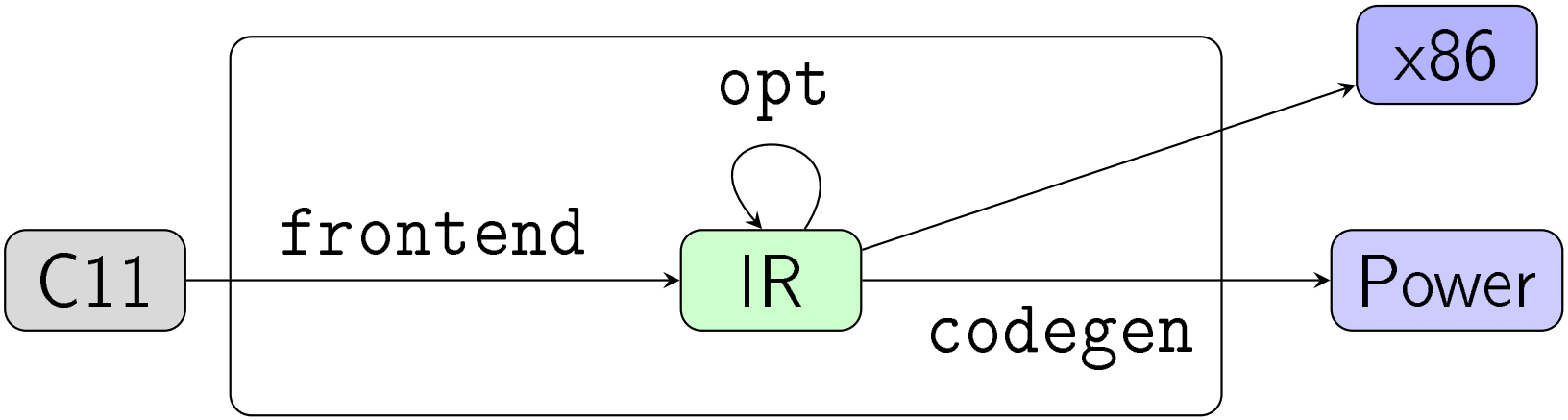
LLVM Compilation



LLVM Concurrency Compilation

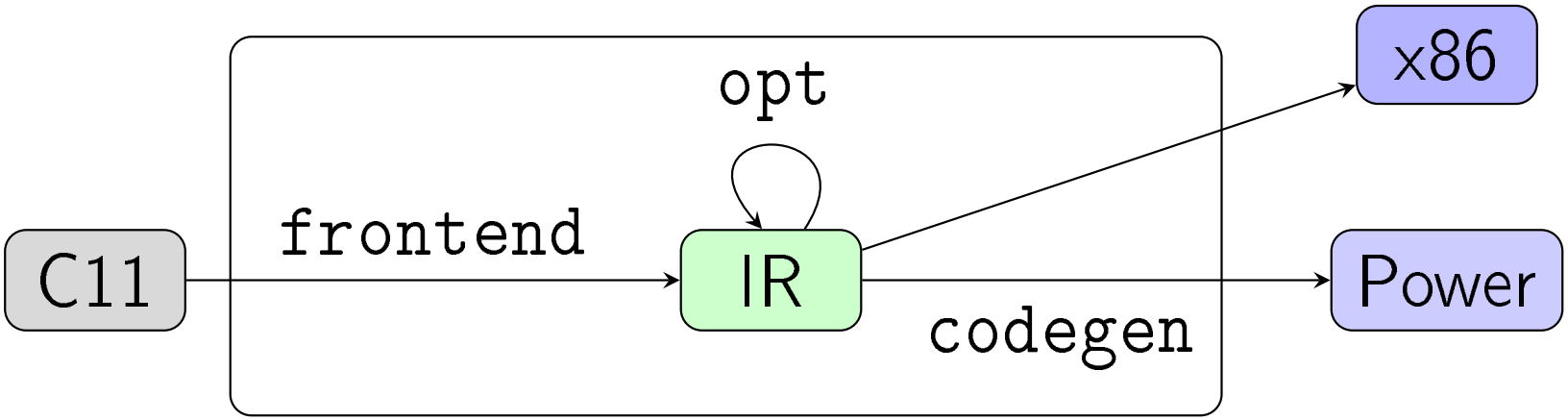


LLVM Concurrency Compilation



formalized **formalized**

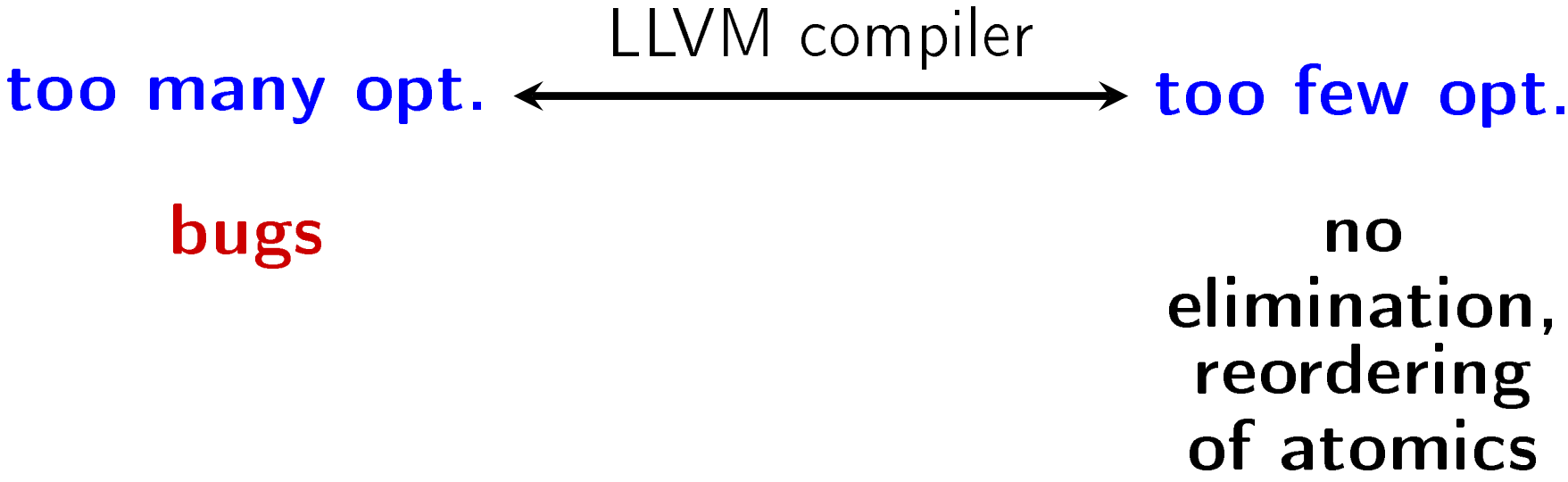
LLVM Concurrency Compilation



formalized **informal** **formalized**

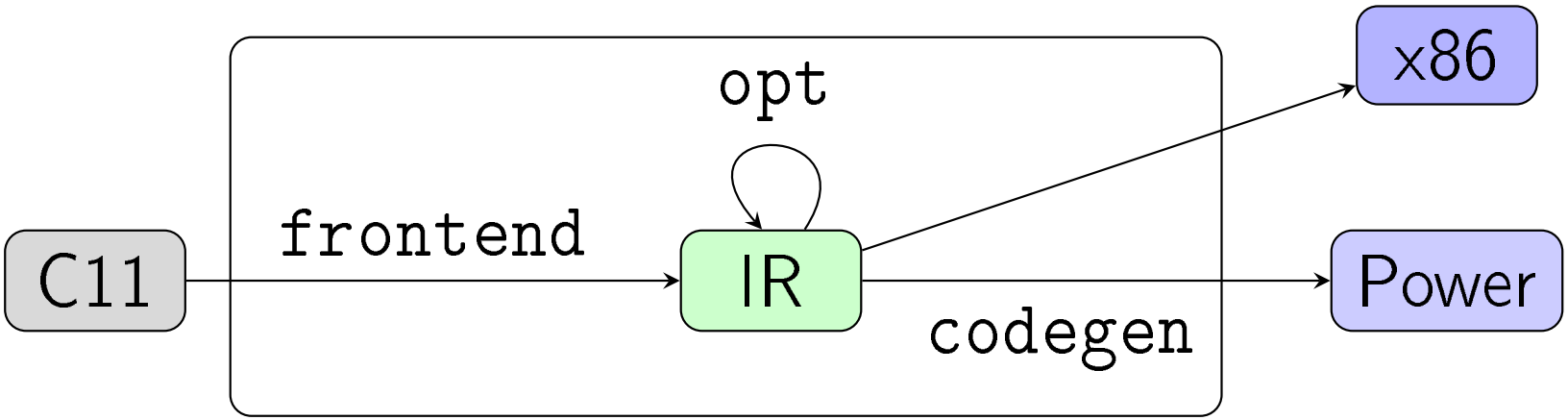
Correctness of the transformations is unclear

Limitation of LLVM Informal Concurrency



Valid opt is removed by over-restriction in bug fix

This Work



formalized **informal** **formalized**

Formalized fragment of LLVM concurrency

(except monotonic/relaxed accesses and fences)

Proved correctness of transformations

Informal LLVM Concurrency

Informal text in *Language Reference Manual*

Frequent references to C11 concurrency

- *"This model is inspired by the C++0x memory model."*
- *"These semantics are borrowed from Java and C++0x, but are somewhat more colloquial."*
- *This is intended to match shared variables in C/C++ ..."*
- ...

Why not adopt C11 concurrency?

Subtle differences

- A program has write-read race on non-atomics
  - C11: the behavior of the program is *undefined*
  - LLVM: *defined* behavior;
    - racy read returns **undef(u)**

$$X = 1; \;\Big\|\; \begin{array}{l} \text{if}(X) \\ \quad t = 4; \\ \text{else} \\ \quad t = 4; \end{array}$$

$t \neq 4$? : C11 ✓ LLVM ✗

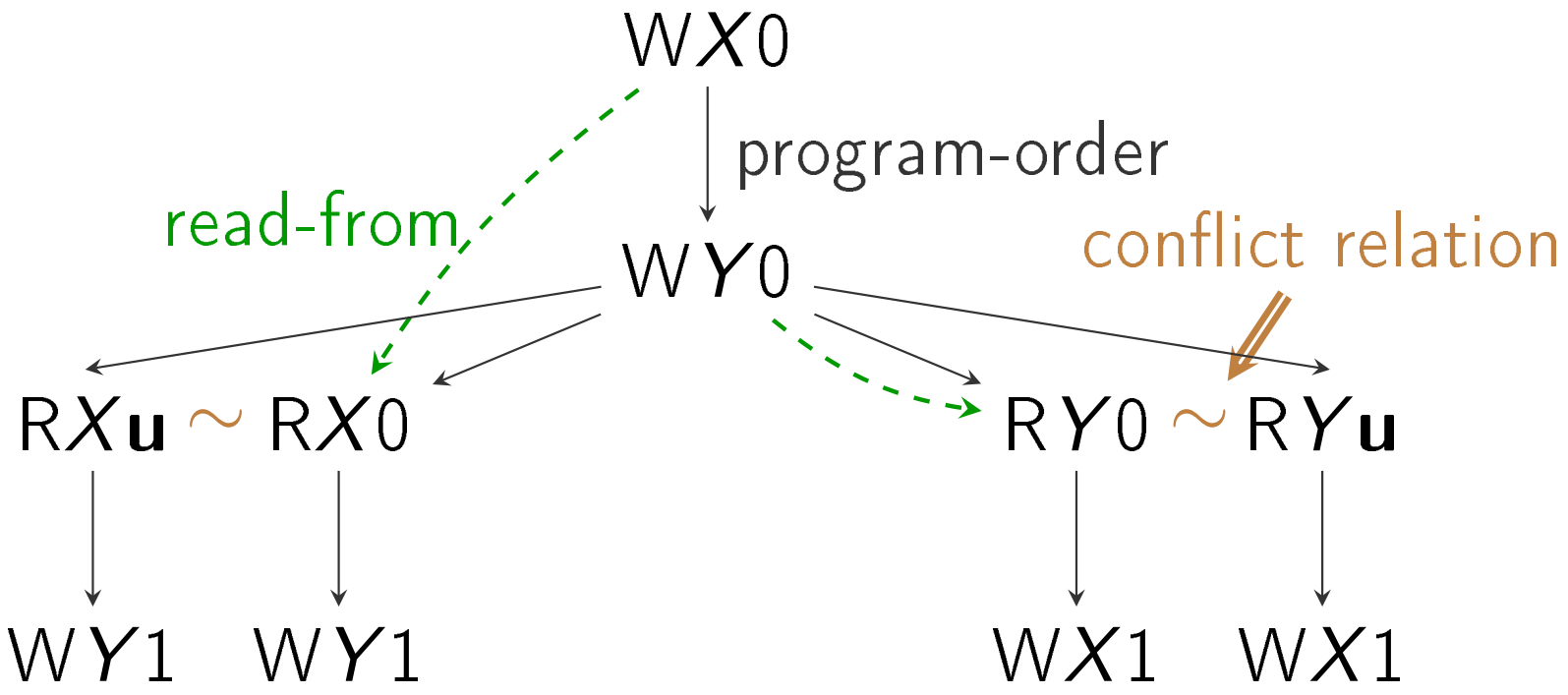
- Set of allowed optimizations are different

Formalization by Event Structure

- Program

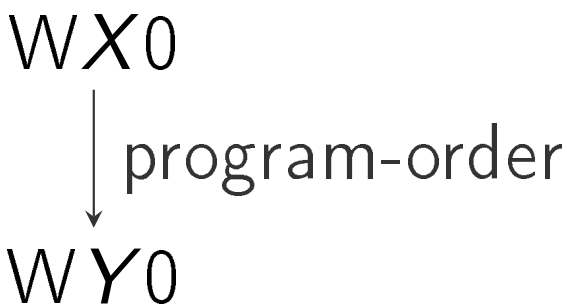
$$\begin{array}{l} int\ X = 0, Y = 0; \\ a = X; \ \| \ b = Y; \\ Y = 1; \ \| \ X = 1; \end{array}$$

- Event Structure



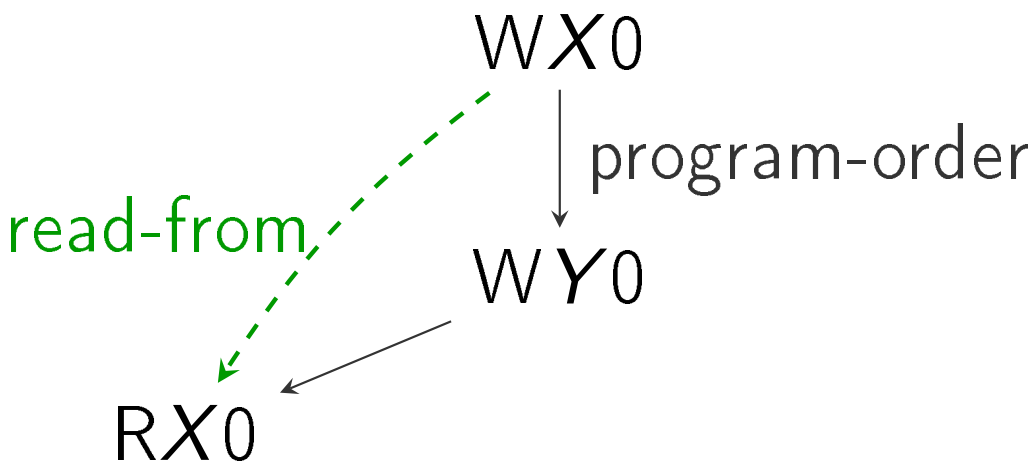
Event Structure Construction

$$\begin{array}{l} int\ X = 0, Y = 0; \\ a = X; \quad \| \quad b = Y; \\ Y = 1; \quad \| \quad X = 1; \end{array}$$



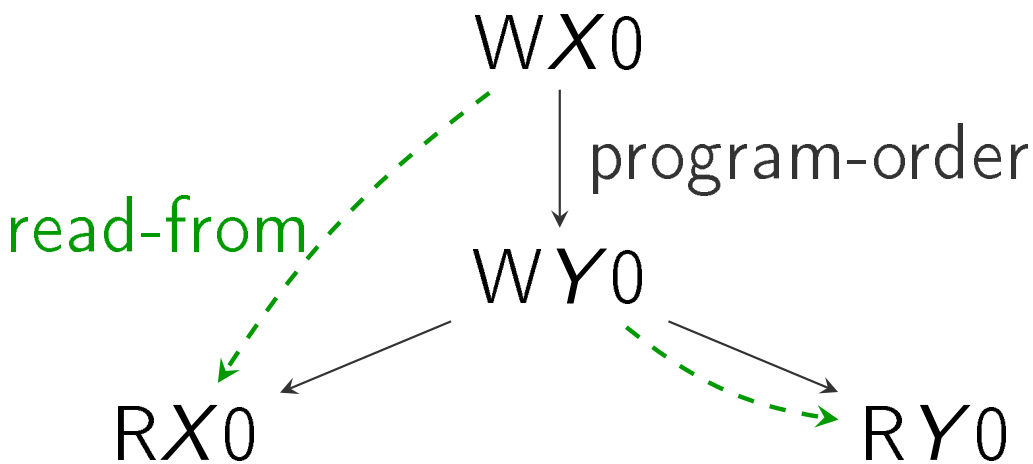
Event Structure Construction

$$\begin{array}{l|l} int\ X = 0, Y = 0; & \\ a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$



Event Structure Construction

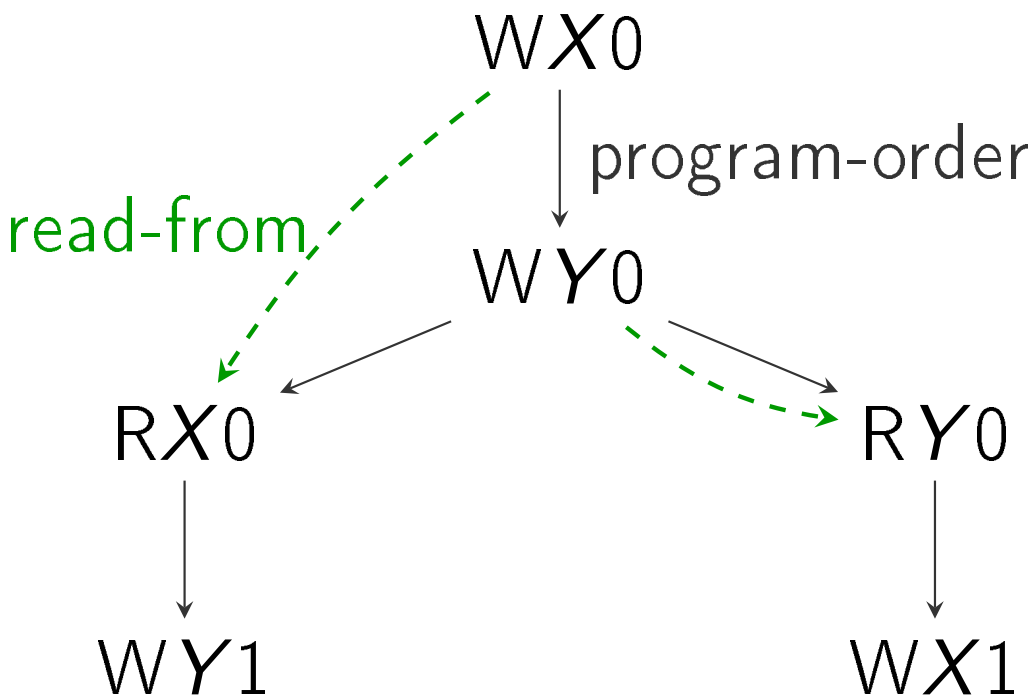
$$
\begin{array}{l|l}
int\ X = 0, Y = 0; & \\
a = X; & b = Y; \\
Y = 1; & X = 1;
\end{array}
$$



Event Structure Construction

$$int\ X = 0, Y = 0;$$

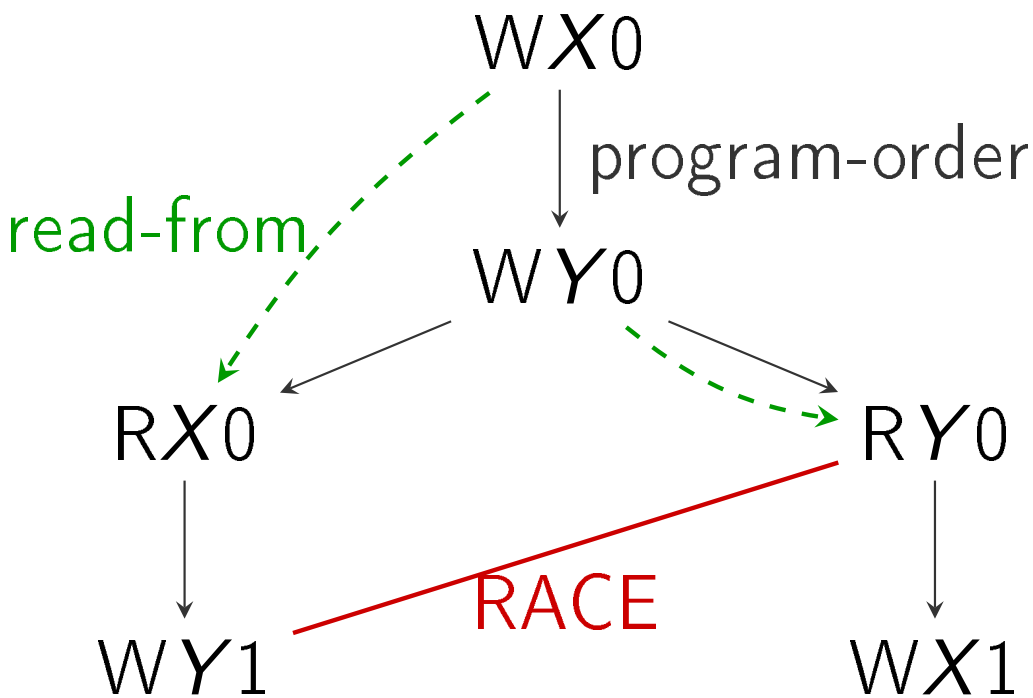
$a = X;$	$\parallel$	$b = Y;$
$Y = 1;$	$\parallel$	$X = 1;$



Event Structure Construction

$$int\ X = 0, Y = 0;$$

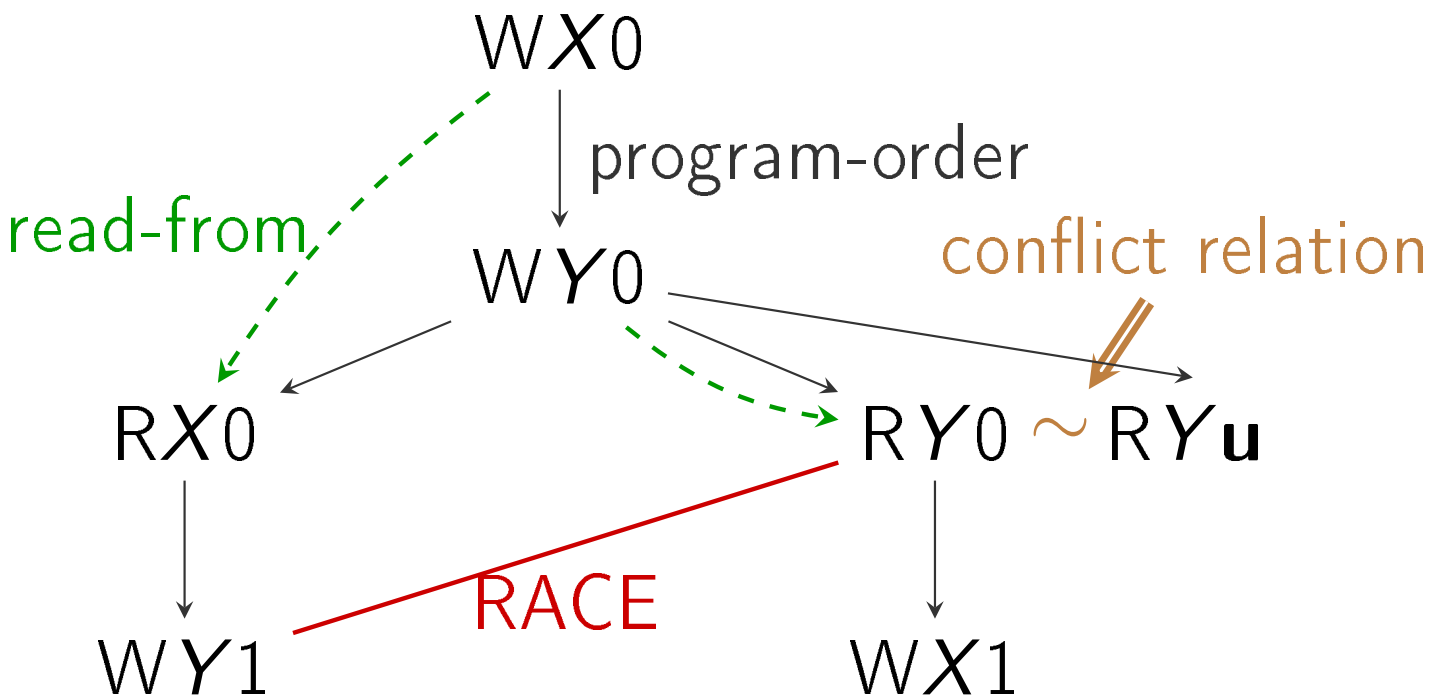
$a = X;$	$b = Y;$
$Y = 1;$	$X = 1;$



Event Structure Construction

$$int\ X = 0, Y = 0;$$

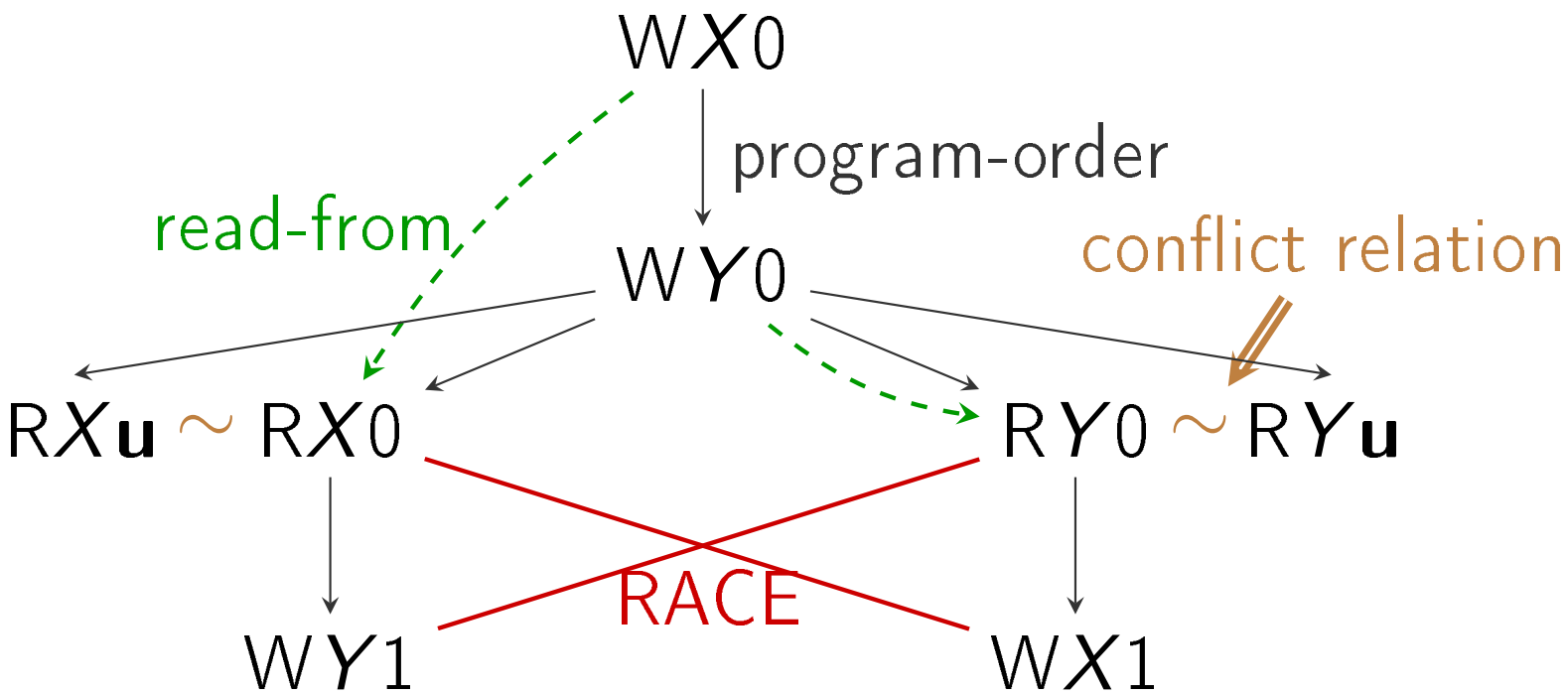
$a = X;$	$b = Y;$
$Y = 1;$	$X = 1;$



Event Structure Construction

$$int\ X = 0, Y = 0;$$

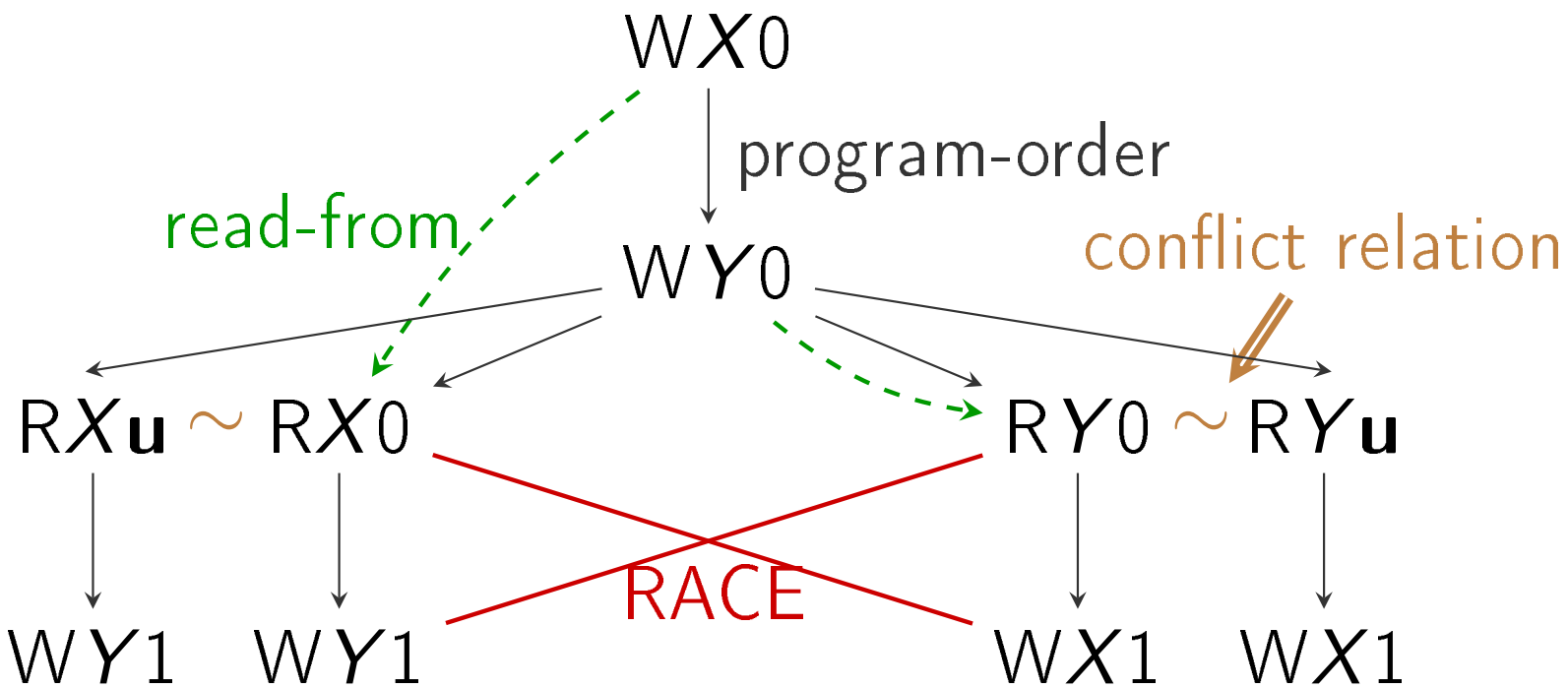
$a = X;$	$b = Y;$
$Y = 1;$	$X = 1;$



Event Structure Construction

$$int\ X = 0, Y = 0;$$

$a = X;$	$b = Y;$
$Y = 1;$	$X = 1;$



Program Behavior

$$
\begin{array}{l||l}
int\ X = 0, Y = 0; & \\
a = X; & b = Y; \\
Y = 1; & X = 1;
\end{array}
$$

$$a = b = 1\ ?\ \checkmark$$

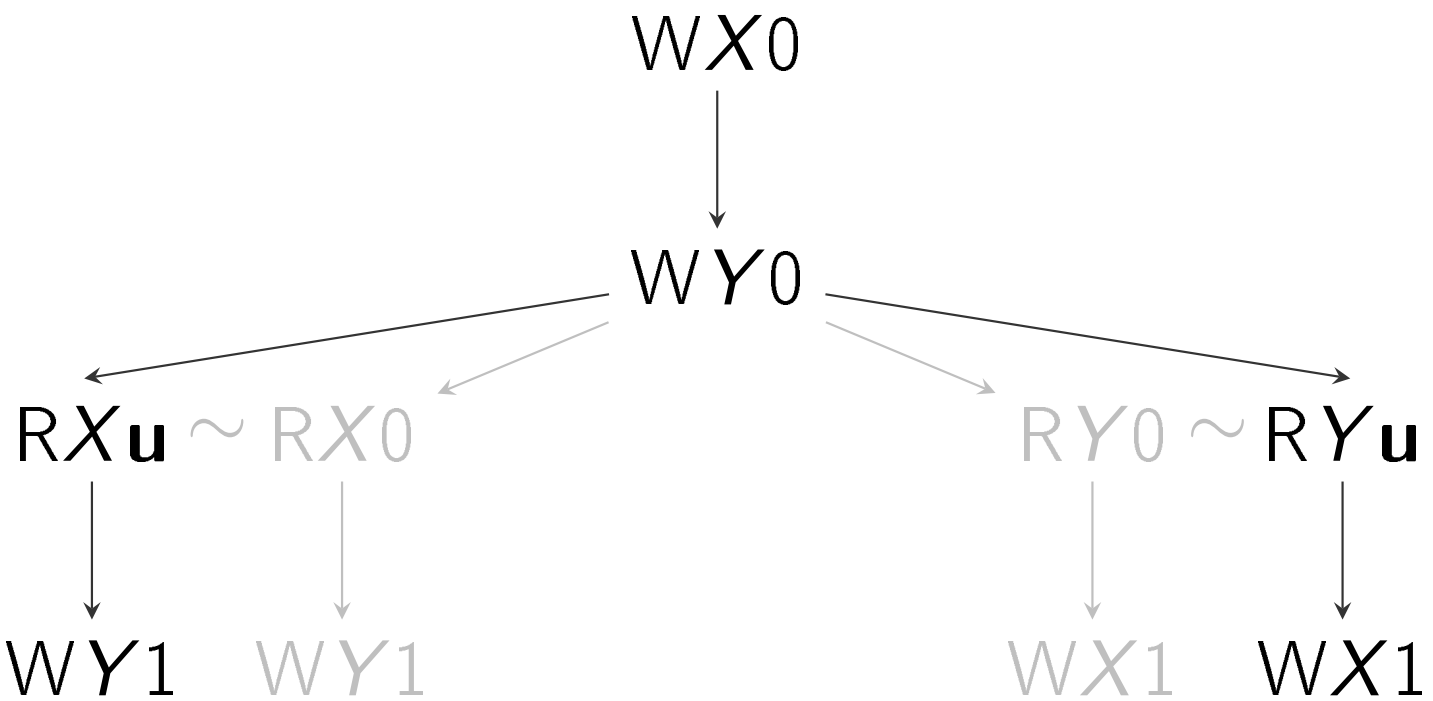
Program Behavior

$$
\begin{array}{c}
int\ X = 0, Y = 0; \\
a = X; \ \| \ b = Y; \\
Y = 1; \ \| \ X = 1; \\
a = b = 1\ ?\ \checkmark
\end{array}
$$

$$
\begin{array}{c}
int\ X = 0, Y = 0; \\
a = X; \ \| \ b = Y; \\
Y = 1; \ \| \ X = 1;
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{c}
int\ X = 0, Y = 0; \\
Y = 1; \ \| \ X = 1; \\
a = X; \ \| \ b = Y;
\end{array}
$$

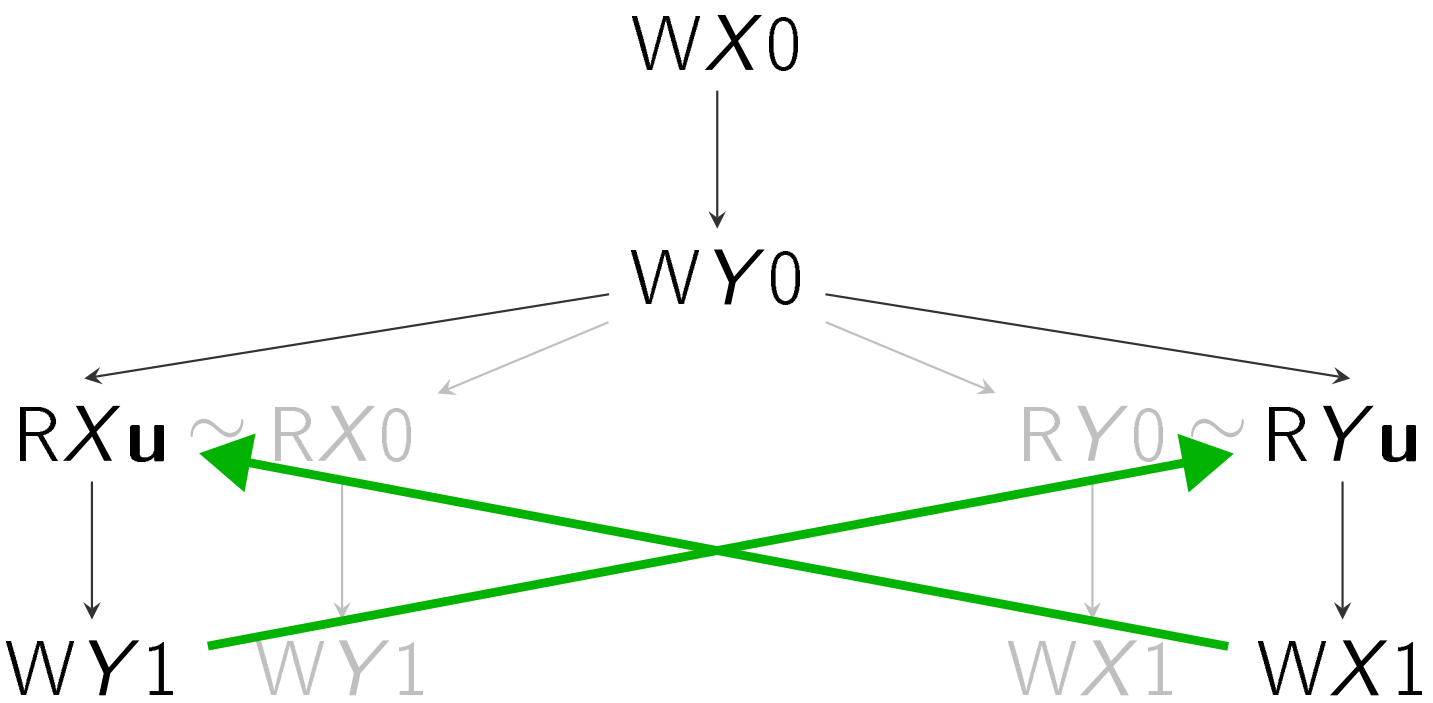
Execution from Event Structure

$$
\begin{array}{l|l}
int\ X = 0, Y = 0; & \\
a = X; & b = Y; \\
Y = 1; & X = 1;
\end{array}
$$



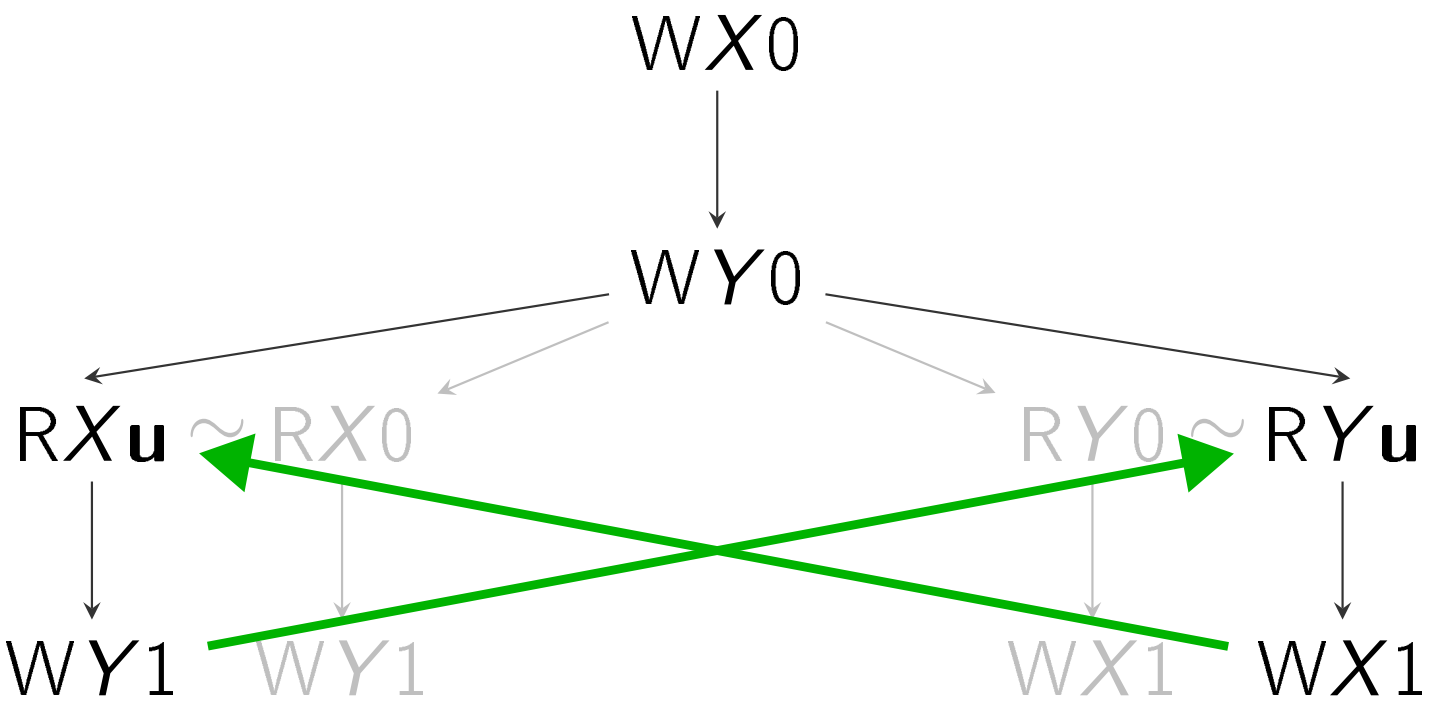
Execution from Event Structure

$$
\begin{array}{l|l}
int\ X = 0, Y = 0; & \\
a = X; & b = Y; \\
Y = 1; & X = 1;
\end{array}
$$



Execution from Event Structure

$$\begin{array}{l|l} int\ X = 0, Y = 0; & \\ a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$



$$a = \mathbf{u} = 1,\ b = \mathbf{u} = 1$$

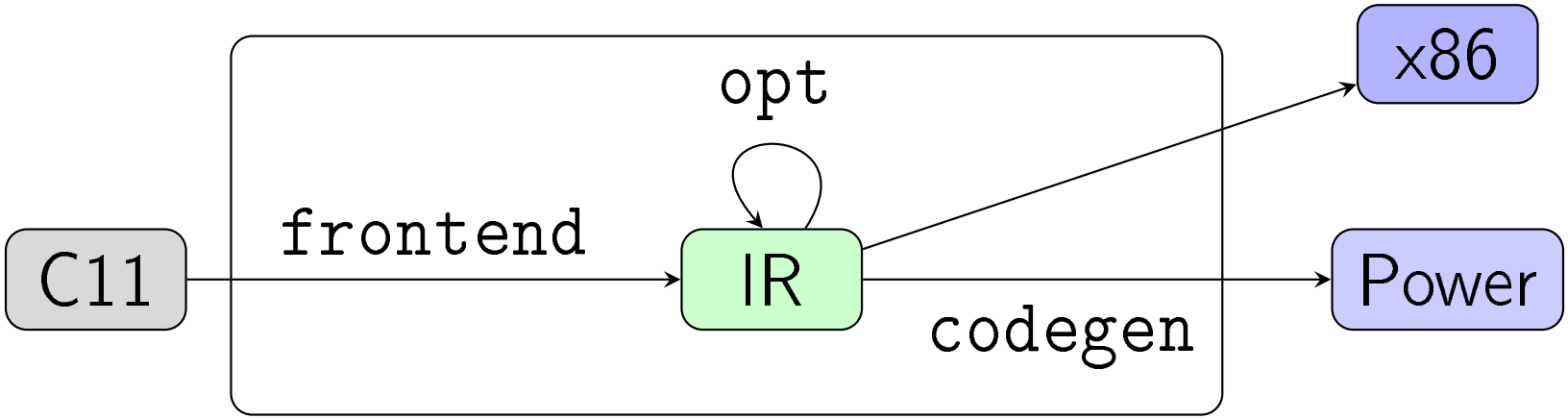
Aspects of Event Structure

- Proposed formalization handles
  - Memory operations: load, store, CAS
  - Memory orders: non-atomic, acquire, release, acquire_release, sequentially consistent (SC)

- Preserves *consistency* at each construction step

- Multiple consistent event structures per program

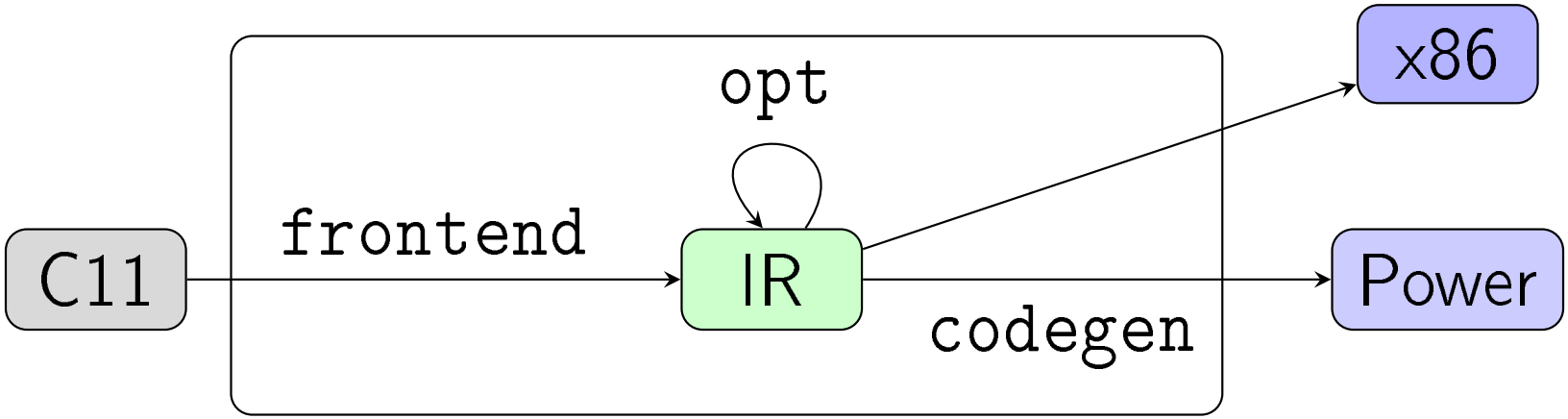
Transformation Correctness



$$\text{Behavior}(P_{tgt}) \subseteq \text{Behavior}(P_{src})$$

Behavior. final values observed in each location

Transformation Correctness



$$\text{Behavior}(P_{tgt}) \subseteq \text{Behavior}(P_{src})$$

Behavior. final values observed in each location

$$\Uparrow$$

$$\text{Behavior}(G_{tgt}) \subseteq \text{Behavior}(G_{src})$$

LLVM performs these eliminations

Adjacent read after read/write elimination

- $a = X_o; b = X_{\mathsf{na}}; \rightsquigarrow a = X_o; b = a;$
- $X_o = v; b = X_{\mathsf{na}}; \rightsquigarrow X_o = v; b = v;$

Adjacent overwritten write elimination

- $X_{\mathsf{na}} = v'; X_{\mathsf{na}} = v; \rightsquigarrow X_{\mathsf{na}} = v;$

Non-adjacent overwritten write elimination

- $X_{\mathsf{na}} = v'; \mathsf{C}; X_{\mathsf{na}} = v; \rightsquigarrow \mathsf{C}; X_{\mathsf{na}} = v;$
  where rel-acq-pair $\notin \mathsf{C}$

LLVM does NOT perform these eliminations

Adjacent read after read/write elimination

- $a = X_{\text{acq}}; b = X_{\text{acq}}; \rightsquigarrow a = X_{\text{acq}}; b = a;$
- $a = X_{\text{sc}}; b = X_{(\text{acq}|\text{sc})}; \rightsquigarrow a = X_{\text{sc}}; b = a;$
- $X_{\text{rel}} = v; b = X_{\text{acq}}; \rightsquigarrow X_{\text{rel}} = v; b = v;$
- $X_{\text{sc}} = v; b = X_{(\text{acq}|\text{sc})}; \rightsquigarrow X_{\text{sc}} = v; b = v;$

Adjacent overwritten write elimination

- $X_{\text{rel}} = v'; X_{\text{rel}} = v; \rightsquigarrow X_{\text{rel}} = v;$
- $X_{(\text{rel}|\text{sc})} = v'; X_{\text{sc}} = v; \rightsquigarrow X_{\text{sc}} = v;$

LLVM does NOT perform these eliminations

Adjacent read after read/write elimination

- $a = X_{\text{acq}}; b = X_{\text{acq}}; \rightsquigarrow a = X_{\text{acq}}; b = a;$
- $a = X_{\text{sc}}; b = X_{(\text{acq}|\text{sc})}; \rightsquigarrow a = X_{\text{sc}}; b = a;$
- $X_{\text{rel}} = v; b = X_{\text{acq}}; \rightsquigarrow X_{\text{rel}} = v; b = v;$
- $X_{\text{sc}} = v; b = X_{(\text{acq}|\text{sc})}; \rightsquigarrow X_{\text{sc}} = v; b = v;$

Adjacent overwritten write elimination

- $X_{\text{rel}} = v'; X_{\text{rel}} = v; \rightsquigarrow X_{\text{rel}} = v;$
- $X_{(\text{rel}|\text{sc})} = v'; X_{\text{sc}} = v; \rightsquigarrow X_{\text{sc}} = v;$

Non-adjacent read after write elimination

- $X_{\text{na}} = v; \text{C}; a = X_{\text{na}}; \rightsquigarrow X_{\text{na}} = v; \text{C}; a = v;$

  where rel-acq-pair $\notin$ C

LLVM performs(✓) these reorderings

$a; b \rightsquigarrow b; a$

$\downarrow a \setminus b \rightarrow$	$(\mathsf{St}\|\mathsf{Ld})_{\mathsf{na}}$	$\mathsf{St}_{\mathsf{rel}}$	$\mathsf{Ld}_{\mathsf{acq}}$	$\mathsf{Ld}_{\mathsf{sc}}$	$\mathsf{U}_{(\mathsf{acq\_rel}\|\mathsf{sc})}$
$(\mathsf{St}\|\mathsf{Ld})_{\mathsf{na}}$	✓	-	✓	✓	-
$\mathsf{St}_{\mathsf{rel}}$	✓	-	-	-	-
$\mathsf{St}_{\mathsf{sc}}$	✓	-	-	-	-
$\mathsf{Ld}_{\mathsf{acq}}$	-	-	-	-	-
$\mathsf{U}_{(\mathsf{acq\_rel}\|\mathsf{sc})}$	-	-	-	-	-

$$X_{\mathsf{rel}} = v; Y_{\mathsf{na}} = v'; \rightsquigarrow Y_{\mathsf{na}} = v'; X_{\mathsf{rel}} = v; \quad ✓$$

LLVM restricts(×) these reorderings

$a; b \rightsquigarrow b; a$

$\downarrow a \setminus b \rightarrow$	$(\mathsf{St}\|\mathsf{Ld})_{\mathsf{na}}$	$\mathsf{St}_{\mathsf{rel}}$	$\mathsf{Ld}_{\mathsf{acq}}$	$\mathsf{Ld}_{\mathsf{sc}}$	$\mathsf{U}_{(\mathsf{acq\_rel}\|\mathsf{sc})}$
$(\mathsf{St}\|\mathsf{Ld})_{\mathsf{na}}$	✓	×	✓	✓	×
$\mathsf{St}_{\mathsf{rel}}$	✓	×	-	-	×
$\mathsf{St}_{\mathsf{sc}}$	✓	×	-	×	×
$\mathsf{Ld}_{\mathsf{acq}}$	×	×	×	×	×
$\mathsf{U}_{(\mathsf{acq\_rel}\|\mathsf{sc})}$	×	×	×	×	×

$$Y_{\mathsf{na}} = v'; X_{\mathsf{rel}} = v; \rightsquigarrow X_{\mathsf{rel}} = v; Y_{\mathsf{na}} = v'; \quad \times$$

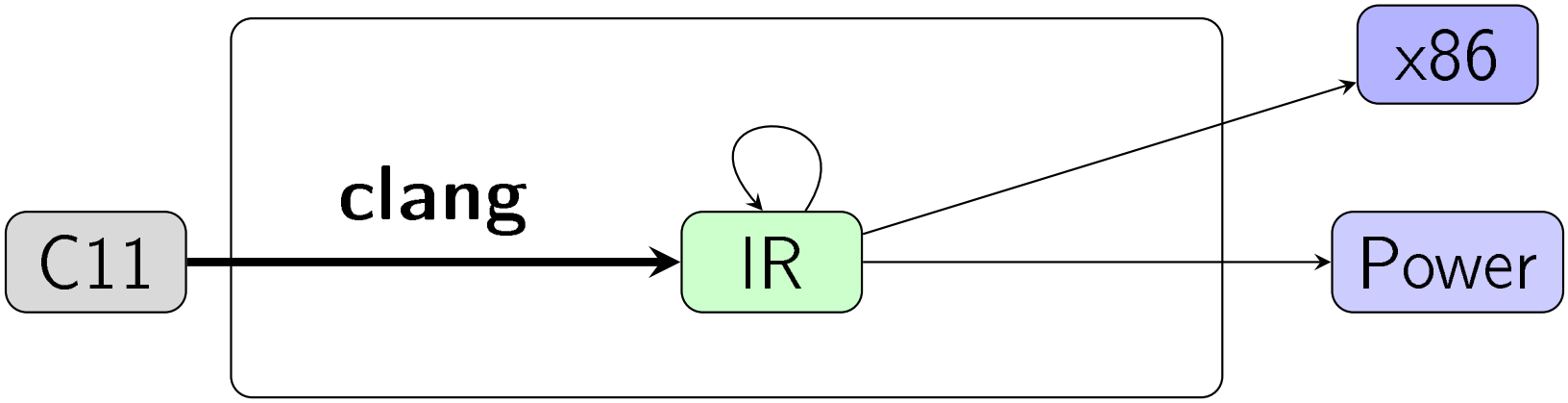
LLVM does NOT perform these reorderings

$a; b \rightsquigarrow b; a$

$\downarrow a \setminus b \rightarrow$	$(\mathsf{St}\|\mathsf{Ld})_{\mathsf{na}}$	$\mathsf{St}_{\mathsf{rel}}$	$\mathsf{Ld}_{\mathsf{acq}}$	$\mathsf{Ld}_{\mathsf{sc}}$	$\mathsf{U}_{(\mathsf{acq\_rel}\|\mathsf{sc})}$
$(\mathsf{St}\|\mathsf{Ld})_{\mathsf{na}}$	✓	×	✓	✓	×
$\mathsf{St}_{\mathsf{rel}}$	✓	×	✓	✓	×
$\mathsf{St}_{\mathsf{sc}}$	✓	×	✓	×	×
$\mathsf{Ld}_{\mathsf{acq}}$	×	×	×	×	×
$\mathsf{U}_{(\mathsf{acq\_rel}\|\mathsf{sc})}$	×	×	×	×	×

$$X_{\mathsf{rel}} = v; t = Y_{\mathsf{acq}}; \rightsquigarrow t = Y_{\mathsf{acq}}; X_{\mathsf{rel}} = v; \quad ✓$$

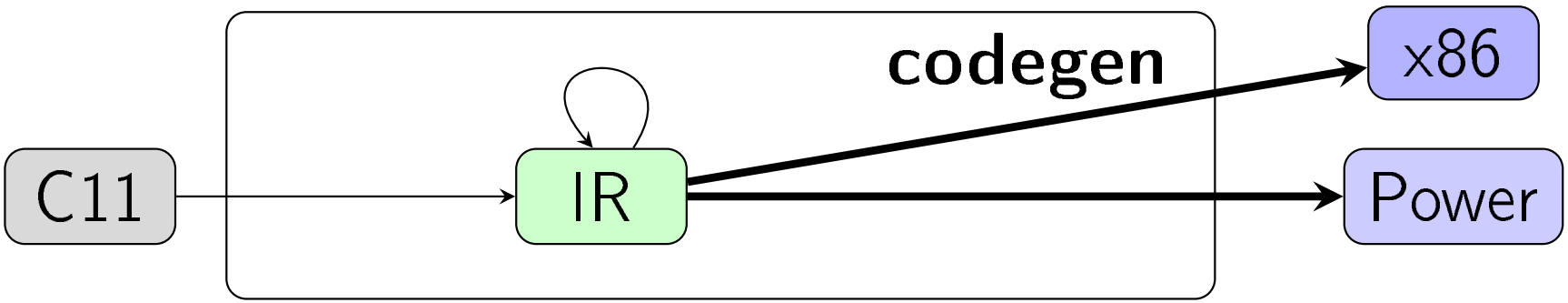
C11 to LLVM Mapping Correctness



- LLVM has operations (Ld/St/CAS) and memory orders (na/rel/acq/acq_rel/SC) similar to C11.

- LLVM model is stronger than C11.

LLVM to Architecture Mapping Correctness



$(\text{LLVM} \rightsquigarrow \text{x86/Power}) = (\text{C11} \rightsquigarrow \text{x86/Power})$

Proved correctness of these mappings

- LLVM to SC
- LLVM to `SPower`

Ensure correctness of LLVM $\rightsquigarrow$ x86/Power
(results from Lahav & Vafeiadis. FM'16)

Event structure construction rules

$$\frac{\begin{array}{c} e \in \mathsf{V} \qquad \forall e'' \in \mathsf{V}.\ e'.\mathsf{id} \neq e''.\mathsf{id} \\ e.\mathsf{code} \xrightarrow{e'.\mathsf{lab}} e'.\mathsf{code} \\ \forall e''.\ \mathsf{po}(e, e'') \implies e'.\mathsf{lab} \neq e''.\mathsf{lab} \end{array}}{\langle \mathsf{V}, \mathsf{po}, \mathsf{rf} \rangle \xrightarrow{e'} \langle \mathsf{V} \cup \{e'\}, (\mathsf{po} \cup \{(e, e')\})^+, \mathsf{rf} \rangle}\ (\textsc{basic})$$

$$\frac{\mathsf{WWrace}(G)}{G \rightsquigarrow G'}\ (\textsc{ww-race}) \qquad \frac{G \xrightarrow{e'} G' \quad e' \notin \mathcal{R}}{G \rightsquigarrow G'}\ (\textsc{non-read})$$

$$\frac{G \xrightarrow{e'} G' \quad e' \in \mathcal{R} \quad e'.\mathsf{rval} = \mathbf{u} \quad \neg G'.\mathsf{hbW}(e')}{G \rightsquigarrow G'}\ (\textsc{r-uninit})$$

$$\frac{G \xrightarrow{e'} G' \quad e' \in \mathcal{R} \quad e'.\mathsf{rval} = \mathbf{u} \quad G'.\mathsf{hbW}(e') \quad \exists w.\ G'.\mathsf{race}(w, e')}{G \rightsquigarrow G'}\ (\textsc{r-race})$$

$$\frac{\begin{array}{c} G \xrightarrow{e'} G' \quad e' \in \mathcal{R} \quad e'.\mathsf{rval} = w.\mathsf{wval} \\ G'.\mathsf{hbW}(e') \quad \neg G'.\mathsf{race}(w, e') \\ G'' = \mathsf{AddRF}(G', w, e') \quad \mathsf{isCons}(G'') \end{array}}{G \rightsquigarrow G''}\ (\textsc{r-norace})$$

What's More in The Paper

Event structure construction rules

Consistency constraints

$$\begin{aligned}\mathsf{isCons}(G) \triangleq\ & \mathsf{irreflexive}(\mathsf{wb}) \wedge \mathsf{irreflexive}(\mathsf{cf};\mathsf{hb}) \\ & \wedge \mathsf{irreflexive}(\mathsf{rf};\mathsf{hb}^{-1};\mathsf{rf}^{-1};\mathsf{cf}) \\ & \wedge \mathsf{acyclic}((\mathsf{hbsc} \cup \mathsf{wb} \cup \mathsf{fr});[\mathsf{SC}])\end{aligned}$$

What's More in The Paper

Event structure construction rules

Consistency constraints

Data race freedom (DRF) theorems

What's More in The Paper

Event structure construction rules

Consistency constraints

Data race freedom (DRF) theorems

More transformations

- Speculative load
- Strengthening memory order of accesses

What's More in The Paper

Event structure construction rules

Consistency constraints

Data race freedom (DRF) theorems

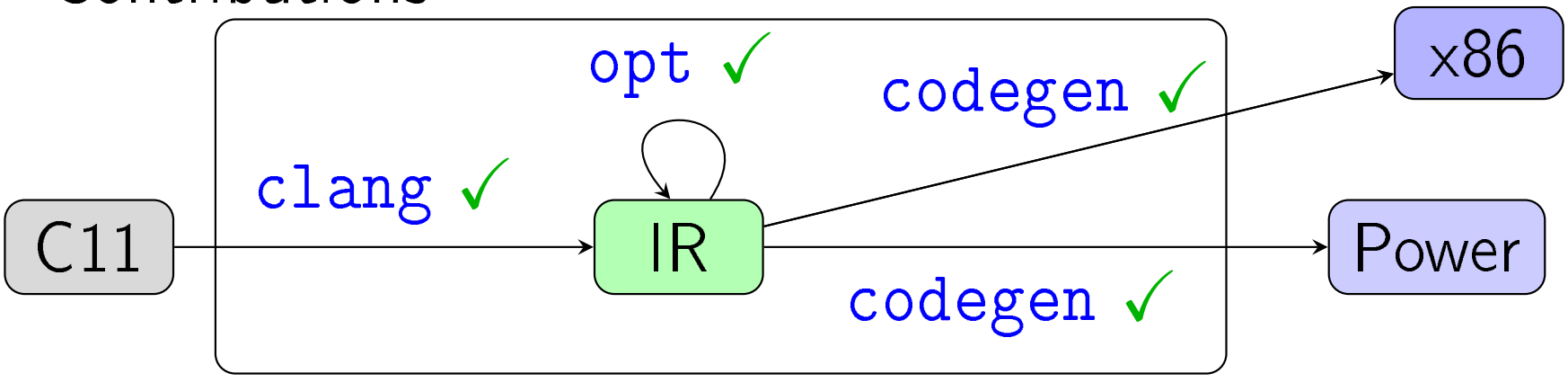
More transformations

- Speculative load
- Strengthening memory order of accesses

Proofs: http://plv.mpi-sws.org/llvmcs/

Conclusions & Future Directions

- Contributions



Formalized ✓

DRF Theorems ✓

- Future: extend the LLVM concurrency model
  - With relaxed accesses and fences
  - Prove/disprove more optimizations
  - Mechanize the formalization

Thank You !

Backup Slides

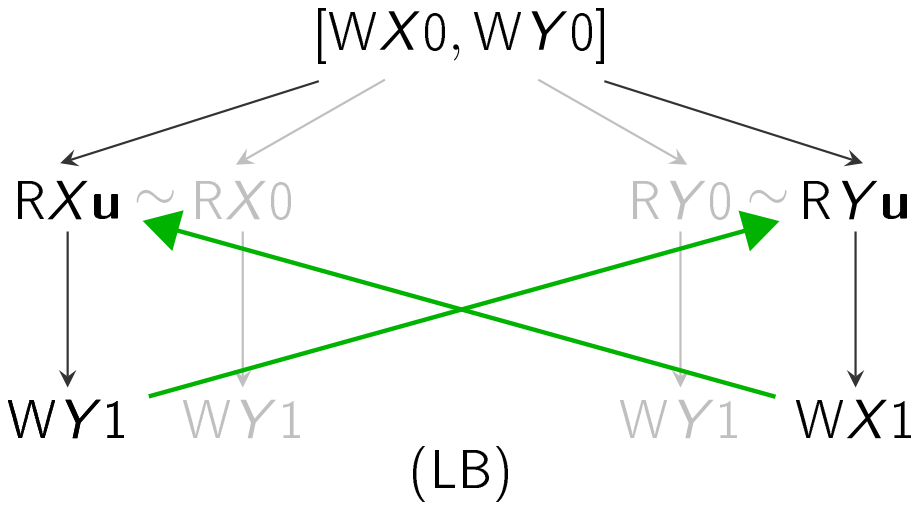
Examples

$$
\begin{array}{c}
int\ X = 0, Y = 0; \\
a = X; \ \| \ b = Y; \\
Y = 1; \ \| \ X = 1; \\
a = b = 1 \ \checkmark
\end{array}
\qquad \text{(LB)}
$$

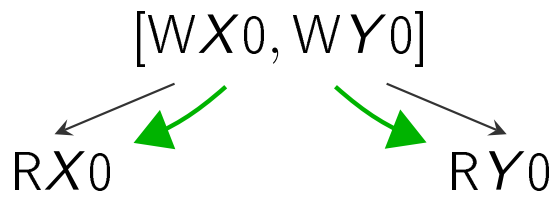
$$
\begin{array}{c}
int\ X = 0, Y = 0; \\
a = X; \ \| \ b = Y; \\
\text{if}(a == 1) \ \| \ \text{if}(b == 1) \\
Y = 1; \ \| \ X = 1; \\
a = b = 1 \ \text{✗}
\end{array}
\qquad \text{(CYC)}
$$

LLVM vs C11

LLVM

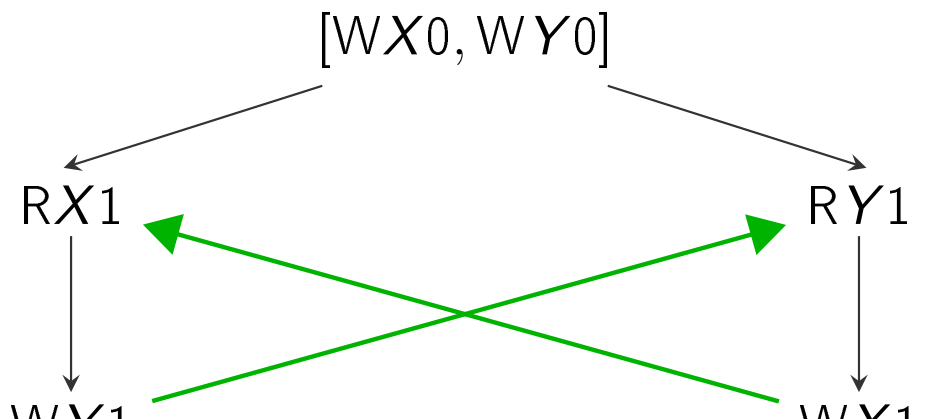


(LB)



(CYC)

C11



(LB & CYC)

LLVM performs speculative load

$$X = 1; \;\Big\|\; \begin{array}{l} \text{if}(\mathit{flag})\{ \\ \quad a = X; \\ \} \end{array} \quad \rightsquigarrow \quad X = 1; \;\Big\|\; \begin{array}{l} t = X; \text{// } \textbf{undef} \\ \text{if}(\mathit{flag})\{ \\ \quad a = t; \\ \} \end{array}$$