Compilation of OCaml memory model to Power

Egor Namakonov, Anton Podkopaev

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS
NATIONAL RESEARCH UNIVERSITY HIGHER SCHOOL OF ECONOMICS

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = c = 1 ?		

An execution result is explained by alternating threads

<table>
<tr><td colspan="3">[x] = [y] = 0</td></tr>
<tr><td>[x] := 1</td><td>a := [x]</td><td>b := [y]</td></tr>
<tr><td></td><td>[y] := 1</td><td>c := [x]</td></tr>
<tr><td colspan="3">a = b = c = 1 ?</td></tr>
</table>

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = c = 1 ?		

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = c = 1 ?		

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = c = 1 ?		

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = c = 1 ?		

An execution result is explained by alternating threads

<table>
<tr><td colspan="3">[x] = [y] = 0</td></tr>
<tr><td>[x] := 1</td><td>a := [x]</td><td>b := [y]</td></tr>
<tr><td></td><td>[y] := 1</td><td>c := [x]</td></tr>
<tr><td colspan="3">a = b = c = 1</td></tr>
</table>

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0 ?		

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0 ?		

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0 ?		

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0 ?		

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0 ?		

An execution result is explained by alternating threads

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0 ?		

An execution result is explained by alternating threads… usually

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0		

C++ allows it due to a (non-atomic) data race

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0		

C++ allows it due to a (non-atomic) data race

[x] = [y] = 0		
$\boxed{\text{[x] := 1}}$	$\boxed{\text{a := [x]}}$	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0		

C++ allows it due to a (non-atomic) data race

Non-atomic accesses

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0		

C++ allows it due to a (non-atomic) data race

Non-atomic accesses

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0		

Standard for Programming Language C++, 6.8.2.1.20:
"Any such data race results in undefined behavior."

No races on atomics

$[x] = [y] = 0$		
$[x]^{rlx} := 1$	$a := [x]^{rlx}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{rlx}$
$a = b = 1, c = 0$?		

No races on atomics

$[x] = [y] = 0$		
$[x]^{rlx} := 1$	$a := [x]^{rlx}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{rlx}$
$a = b = 1, c = 0$?		

Constants

Defined in header <atomic>

Value
`memory_order_relaxed`

No races on atomics but the outcome is still allowed

$[x] = [y] = 0$		
$[x]^{rlx} := 1$	$a := [x]^{rlx}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{rlx}$
$a = b = 1, c = 0$		

C++ **memory model**

$[x] = [y] = 0$		
$[x]^{rlx} := 1$	$a := [x]^{rlx}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{rlx}$

C++ **memory model**

$\llbracket$

$[x] = [y] = 0$		
$[x]^{\mathrm{rlx}} := 1$	$a := [x]^{\mathrm{rlx}}$	$b := [y]^{\mathrm{rlx}}$
	$[y]^{\mathrm{rlx}} := 1$	$c := [x]^{\mathrm{rlx}}$

$\rrbracket$

$$= \{ \ldots, (a=b=c=1), \ldots (a=b=1, c=0), \ldots \}$$

C++ memory model is **weak**

⟦

$[x] = [y] = 0$		
$[x]^{rlx} := 1$	$a := [x]^{rlx}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{rlx}$

⟧

$$= \{ \ldots, (a=b=c=1), \ldots \boxed{(a=b=1, c=0),} \ldots \}$$

C++ memory model is **weak** as it allows optimizations

$[x] = [y] = 0$		
$[x]^{rlx} := 1$	$a := [x]^{rlx}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{rlx}$

$$= \{\ \ldots,\ (a=b=c=1),\ \ldots\ (a=b=1,\ c=0),\ \ldots\ \}$$

C++ memory model is **weak** as it allows optimizations

$$\Bigg[\!\!\Bigg[$$

	$[x] = [y] = 0$	
$[x]^{rlx} := 1$	$a := [x]^{rlx}$	$b :$ … ^{rlx}
	$[y]^{rlx} := 1$	$c := [x]^{rlx}$

$$\Bigg]\!\!\Bigg]$$

$$= \{\ \ldots,\ (a{=}b{=}c{=}1),\ \ldots\ (a{=}b{=}1,\ c{=}0),\ \ldots\ \}$$

C++ memory model is **weak** as it allows optimizations

$\llbracket$

$[x] = [y] = 0$		
$[x]^{rlx} := 1$	$a := [x]^{rlx}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{rlx}$

$\rrbracket$

$$= \{ \ldots, (a=b=c=1), \ldots\ (a=b=1, c=0), \ldots \}$$

Weak behavior can be controlled with access modes

[x] = [y] = 0		
$[x]^{sc} := 1$	$a := [x]^{sc}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{sc}$

Weak behavior can be controlled with access modes

$[x] = [y] = 0$		
$[x]^{sc} := 1$	$a := [x]^{sc}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{sc}$

Weak behavior can be controlled with access modes but the effect is not obvious

$[x] = [y] = 0$		
$[x]^{sc} := 1$	$a := [x]^{sc}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{sc}$
$a = b = 1, c = 0$		

POWER

C++ solution: strengthen access mode everywhere

[x] = [y] = 0		
$[x]^{sc} := 1$	$a := [x]^{sc}$	$b := [y]^{sc}$
	$[y]^{sc} := 1$	$c := [x]^{sc}$
~~a = b = 1, c = 0~~		

OCaml MM: reasonable rules for racy programs

$[x] = [y] = 0$		
$[x]^{at} := 1$	$a := [x]^{at}$	$b := [y]^{na}$
	$[y]^{na} := 1$	$c := [x]^{at}$

OCaml MM: reasonable rules for racy programs

[x] = [y] = 0		
$[x]^{at}$:= 1	a := $[x]^{at}$	b := $[y]^{na}$
	$[y]^{na}$:= 1	c := $[x]^{at}$

OCaml MM: reasonable rules for racy programs

$[x] = [y] = 0$		
$[x]^{at} := 1$	$a := [x]^{at}$	$b := [y]^{na}$
	$[y]^{na} := 1$	$c := [x]^{at}$

OCaml MM: reasonable rules for racy programs

$[x] = [y] = 0$		
$[x]^{at} := 1$	$a := [x]^{at}$	$\boxed{b := [y]^{na}}$
	$\boxed{[y]^{na} := 1}$	$c := [x]^{at}$

OCaml MM: reasonable rules for racy programs

$[x] = [y] = 0$		
$[x]^{at} := 1$	$a := [x]^{at}$	$\boxed{b := [y]^{na}}$
	$\boxed{[y]^{na} := 1}$	$c := [x]^{at}$

Local data race freedom:
result of reading x doesn't depend on the race on y

OCaml MM: reasonable rules for racy programs

$[x] = [y] = 0$		
$[x]^{at} := 1$	$a := [x]^{at}$	$\boxed{b := [y]^{na}}$
	$\boxed{[y]^{na} := 1}$	$c := [x]^{at}$

Local data race freedom:
result of reading x doesn't depend on the race on y

OCaml MM: reasonable rules for racy programs

$[x] = [y] = 0$		
$[x]^{at} := 1$	$a := [x]^{at}$	$\boxed{b := [y]^{na}}$
	$\boxed{[y]^{na} := 1}$	$c := [x]^{at}$

Local data race freedom:
result of reading x doesn't depend on the race on y

OCaml MM: reasonable rules for racy programs

$[x] = [y] = 0$		
$[x]^{at} := 1$	$a := [x]^{at}$	$\boxed{b := [y]^{na}}$
	$\boxed{[y]^{na} := 1}$	$c := [x]^{at}$

Local data race freedom:
result of reading x doesn't depend on the race on y

OCaml MM: reasonable rules for racy programs

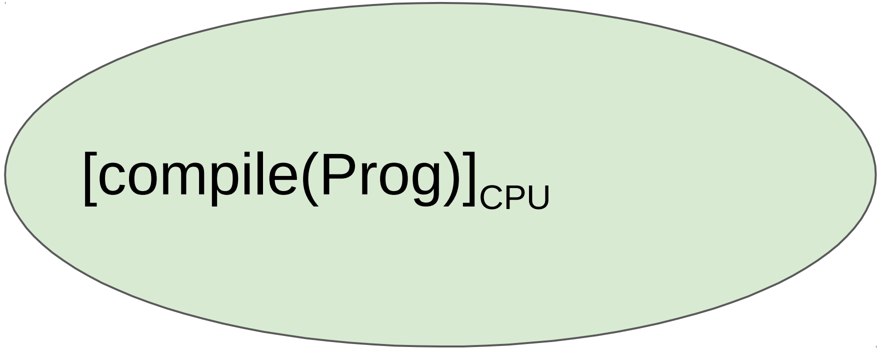
$[x] = [y] = 0$		
$[x]^{at} := 1$	$a := [x]^{at}$	$\boxed{b := [y]^{na}}$
	$\boxed{[y]^{na} := 1}$	$c := [x]^{at}$
~~$a = b = 1, c = 0$~~		

Local data race freedom:
result of reading x doesn't depend on the race on y

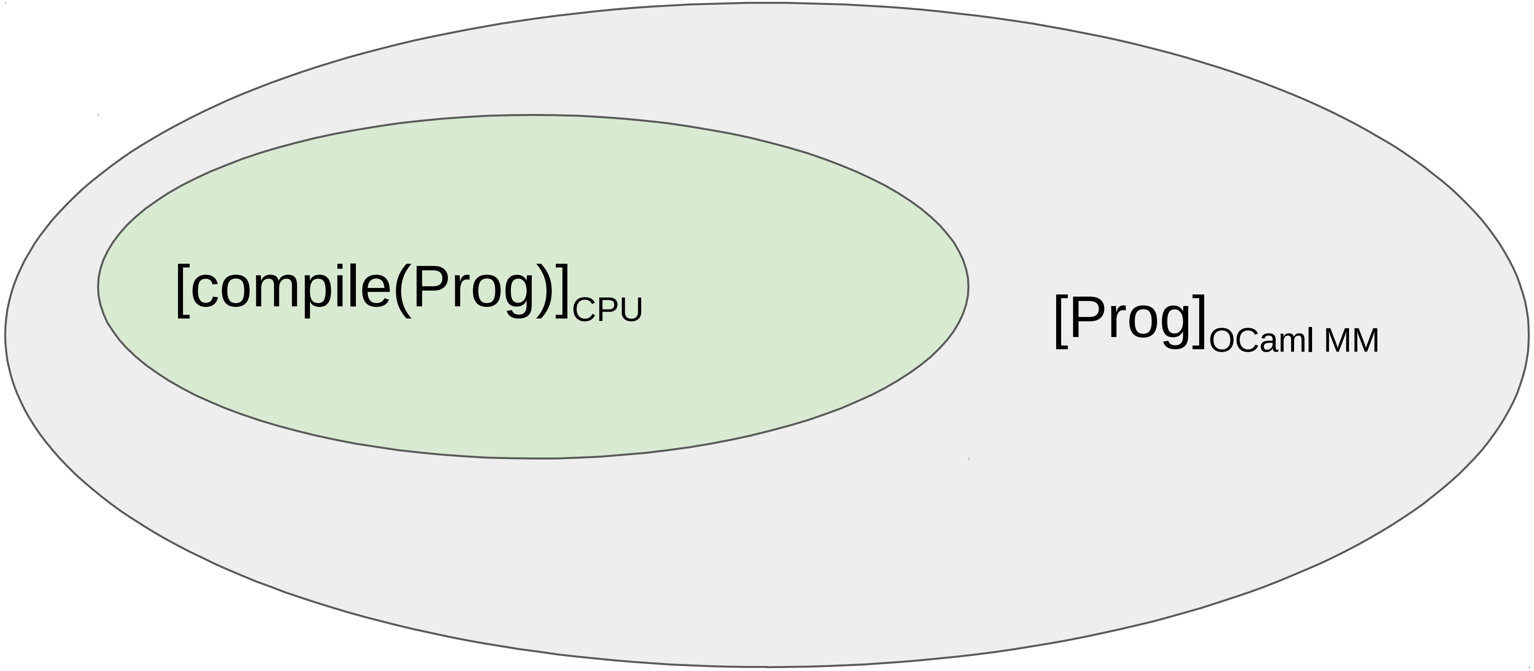
OCaml MM guarantees should be implemented

<table>
<tr><td colspan="3">$[x] = [y] = 0$</td></tr>
<tr><td>$[x]^{sc} := 1$</td><td>$a := [x]^{sc}$</td><td>$b := [y]^{rlx}$</td></tr>
<tr><td></td><td>$[y]^{rlx} := 1$</td><td>$c := [x]^{sc}$</td></tr>
<tr><td colspan="3">$a = b = 1, c = 0$</td></tr>
</table>

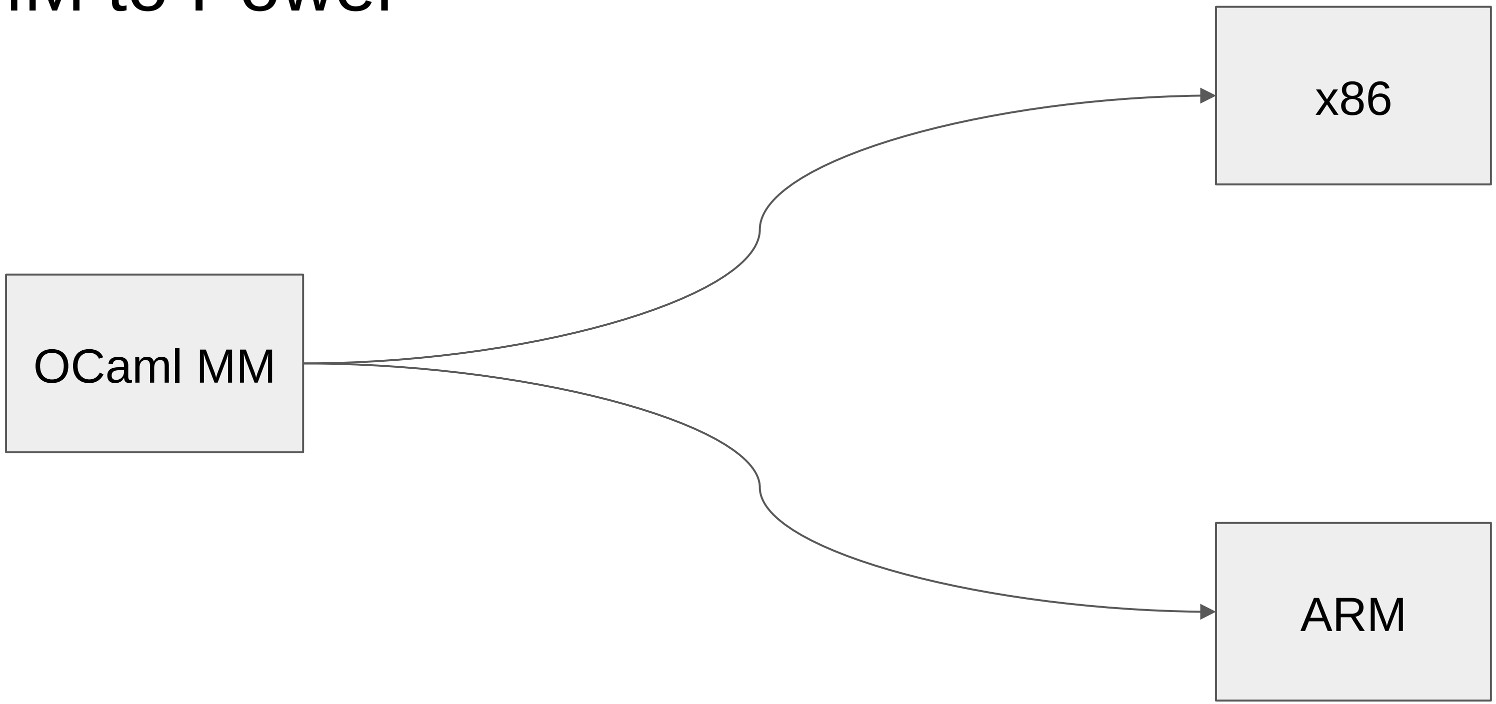
OCaml MM guarantees should be implemented



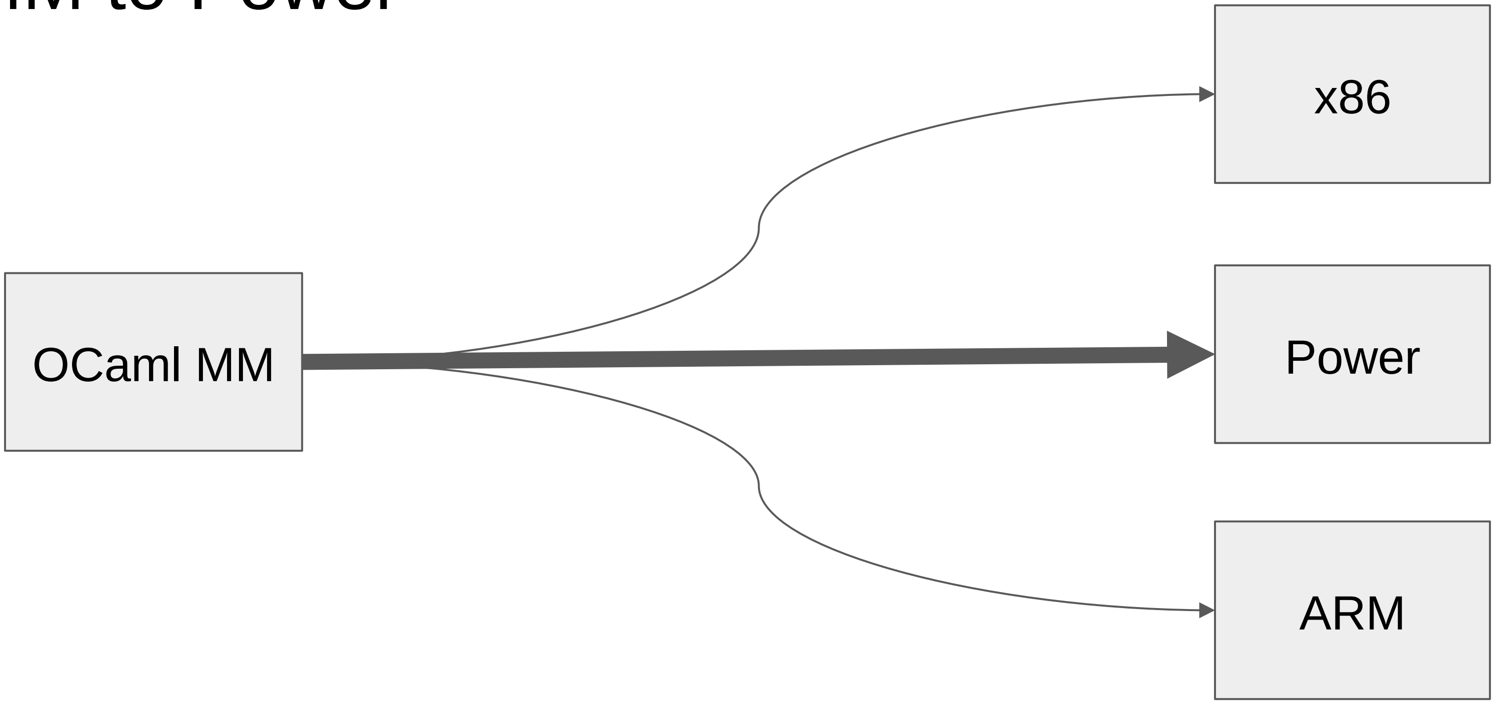
OCaml MM guarantees should be implemented by providing a correct compilation scheme



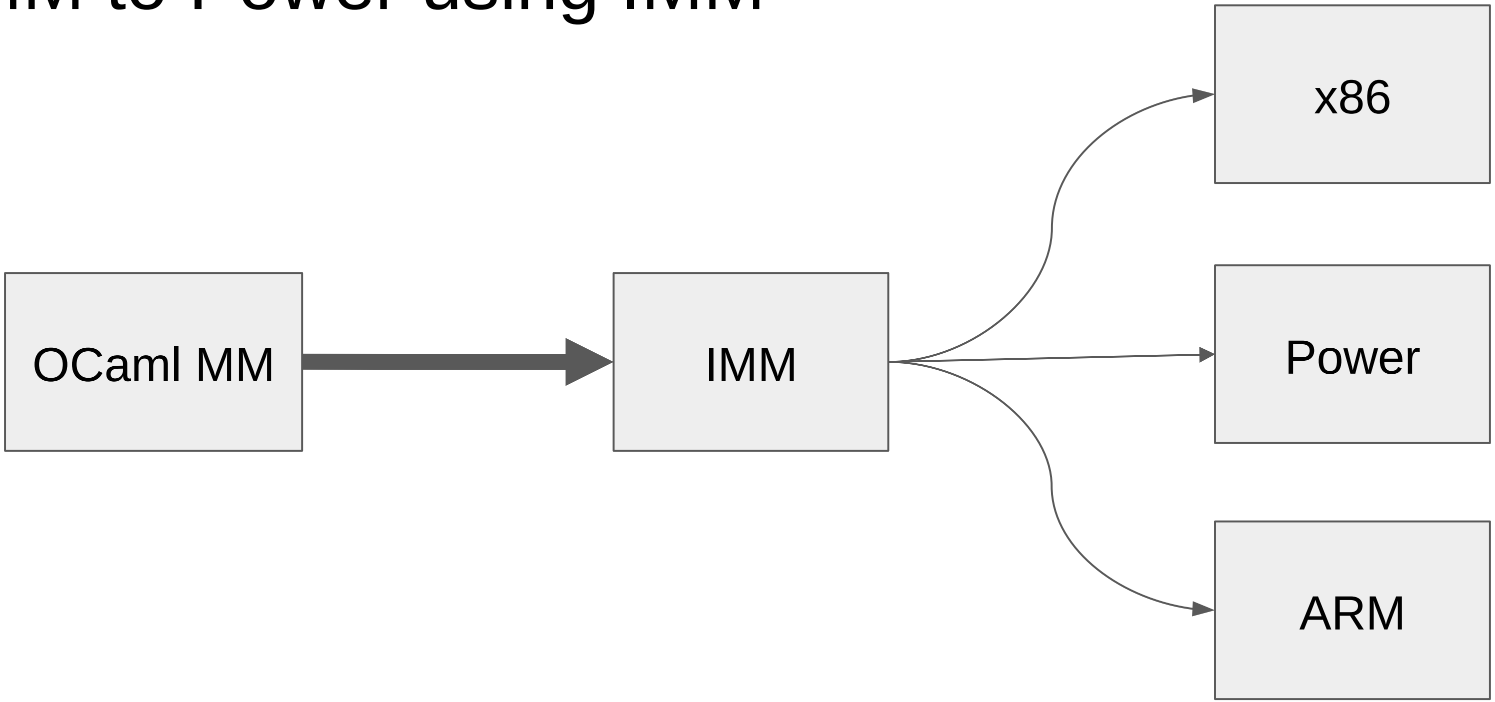
We’ve proved compilation correctness from OCaml MM to Power



We’ve proved compilation correctness from OCaml MM to Power



We've proved compilation correctness from OCaml MM to Power using IMM

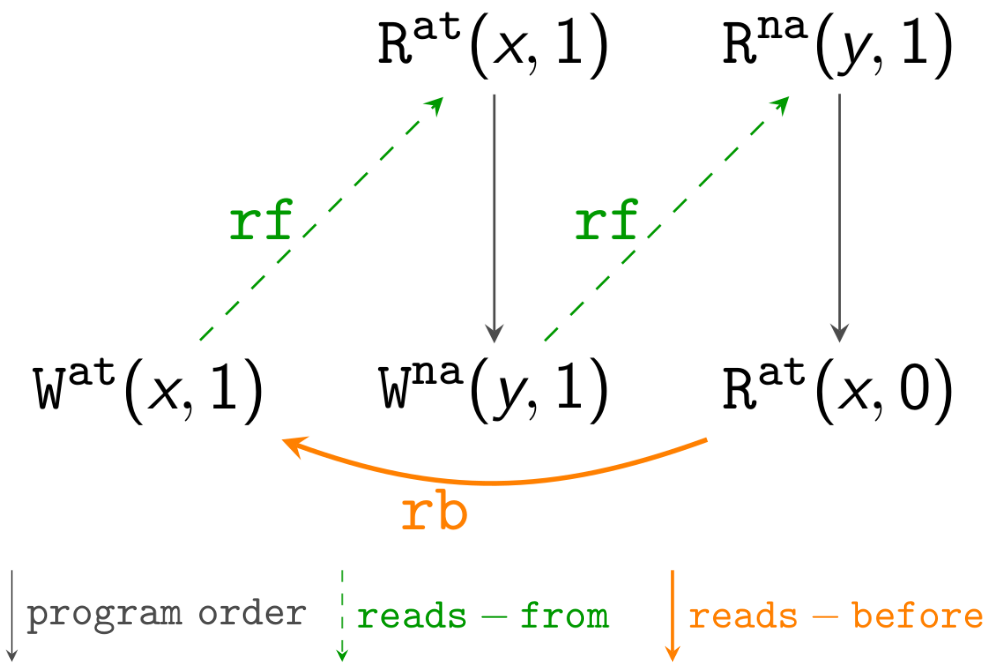


Another execution representation is needed

[x] = [y] = 0		
[x] := 1	a := [x]	b := [y]
	[y] := 1	c := [x]
a = b = 1, c = 0		

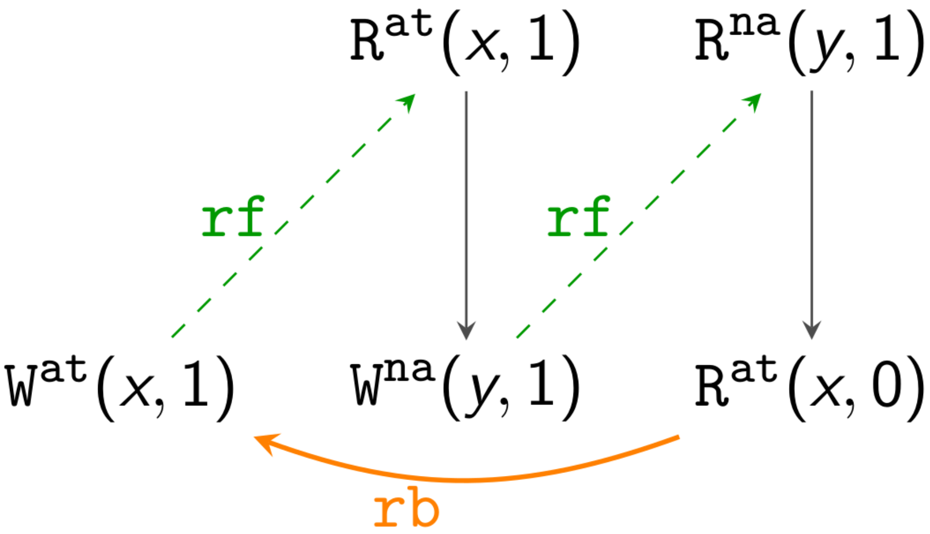
Consider the execution as a graph

[x] = [y] = 0		
$[x]^{at} := 1$	$a := [x]^{at}$	$b := [y]^{na}$
	$[y]^{na} := 1$	$c := [x]^{at}$
a = b = 1, c = 0		



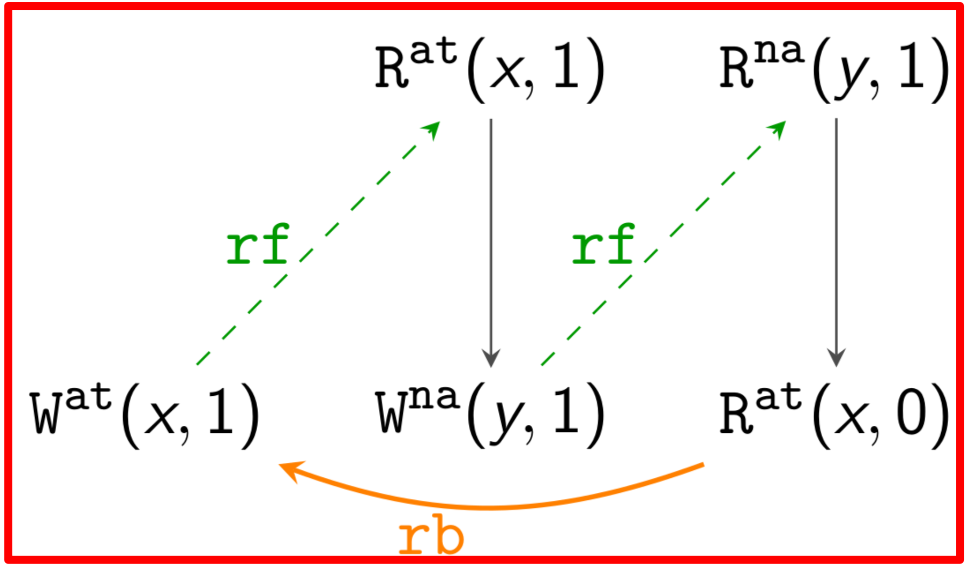
A permission of execution is determined by its graph

[x] = [y] = 0		
$[x]^{at} := 1$	$a := [x]^{at}$	$b := [y]^{na}$
	$[y]^{na} := 1$	$c := [x]^{at}$



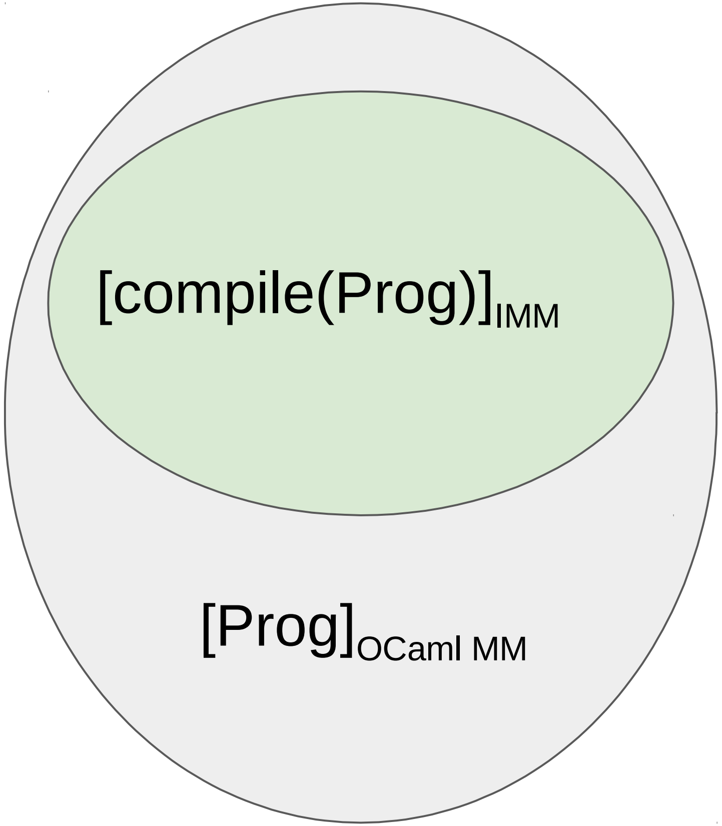
A permission of execution is determined by its graph

[x] = [y] = 0		
$[x]^{at} := 1$	$a := [x]^{at}$	$b := [y]^{na}$
	$[y]^{na} := 1$	$c := [x]^{at}$
OCaml MM: ~~a = b = 1, c = 0~~		

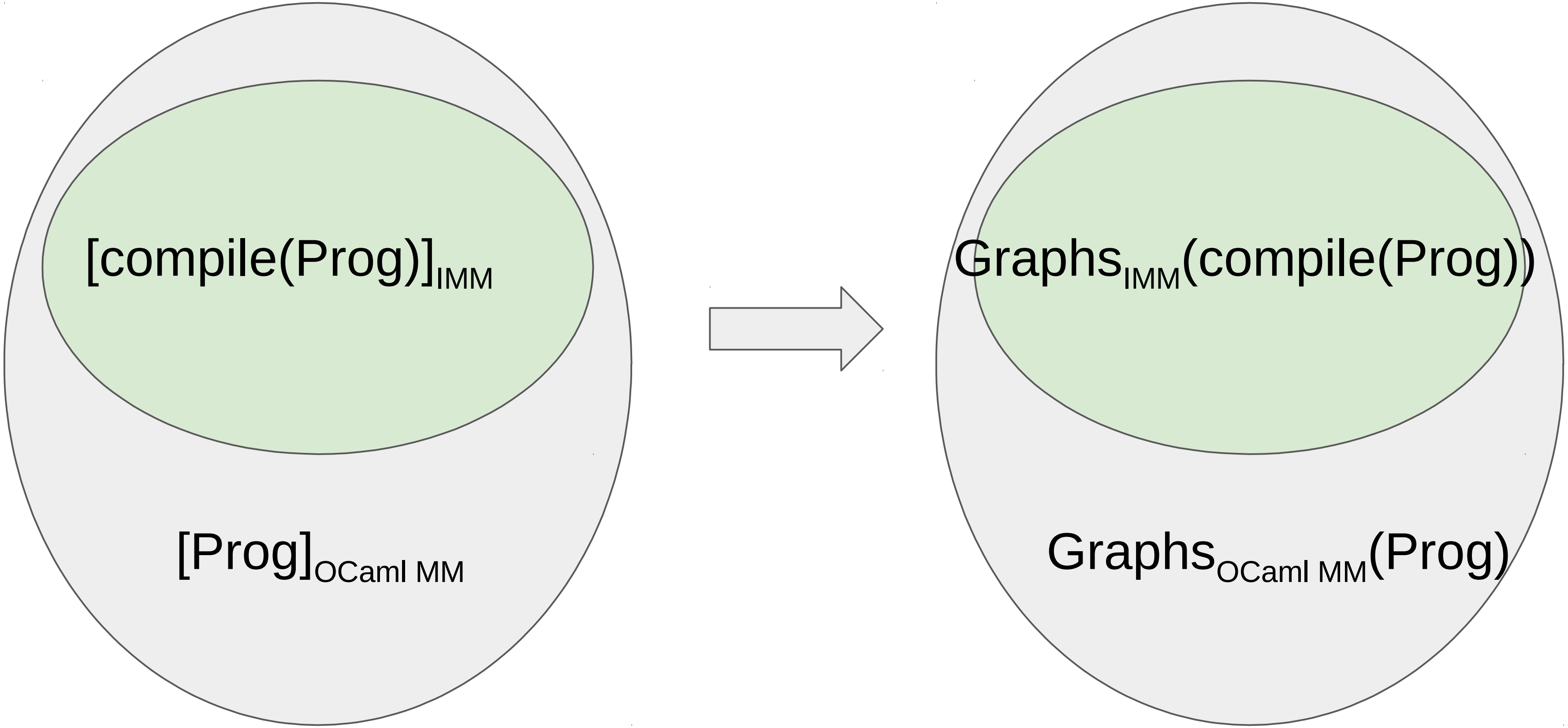


OMM: no cycles made of **po**, **rf** and **rb**

Compilation correctness in terms of graphs



Compilation correctness in terms of graphs



The identity compilation scheme won’t work

compile(Prog) = [na $\rightarrow$ rlx, at $\rightarrow$ sc]Prog

The identity compilation scheme won’t work

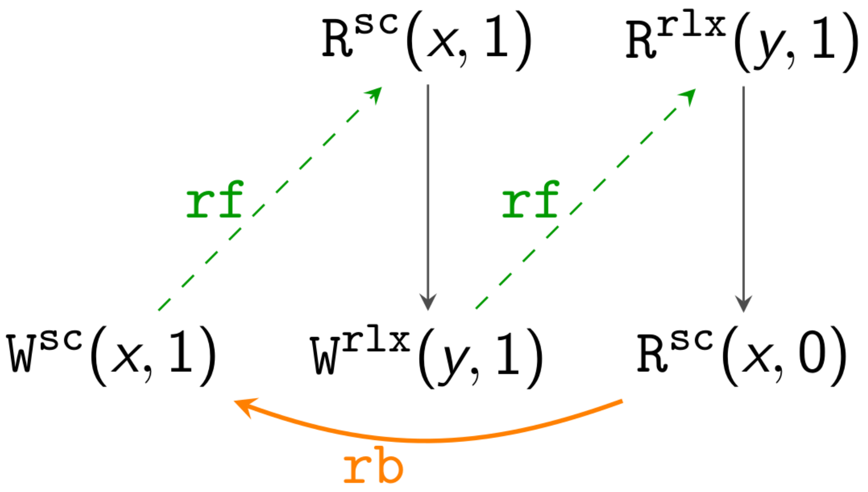
compile(Prog) = [na → rlx, at → sc]Prog

$[x] = [y] = 0$		
$[x]^{sc} := 1$	$a := [x]^{sc}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{sc}$

The identity compilation scheme won't work

compile(Prog) = [na → rlx, at → sc]Prog

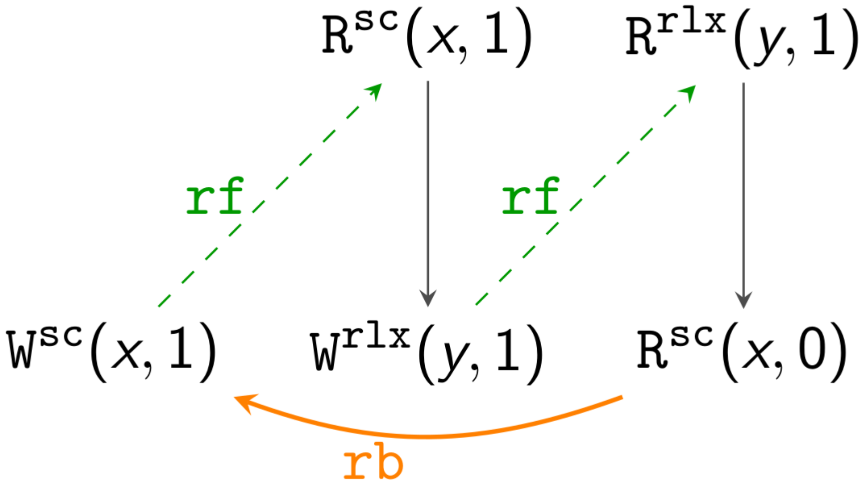
[x] = [y] = 0		
$[x]^{sc} := 1$	$a := [x]^{sc}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{sc}$



The identity compilation scheme won't work

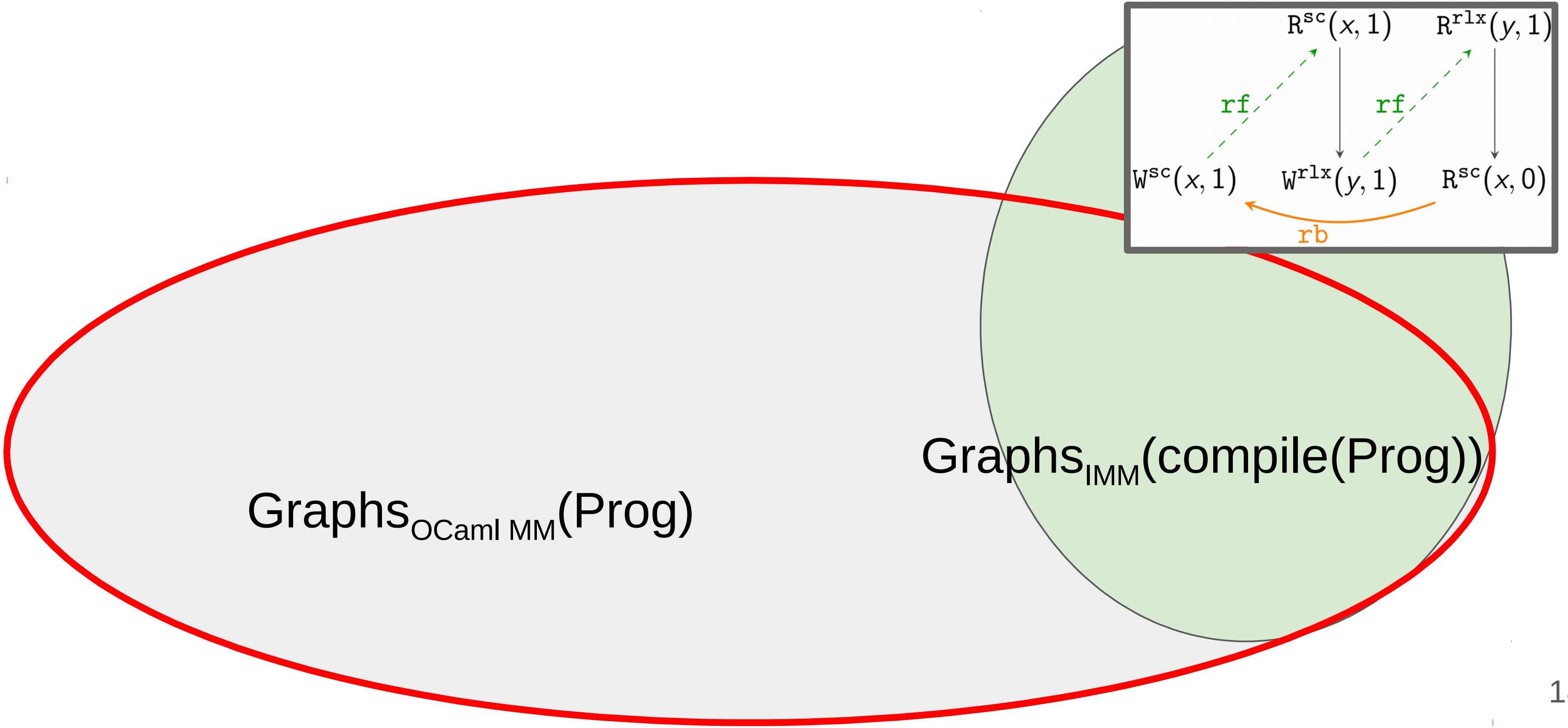
compile(Prog) = [na → rlx, at → sc]Prog

[x] = [y] = 0		
$[x]^{sc} := 1$	$a := [x]^{sc}$	$b := [y]^{rlx}$
	$[y]^{rlx} := 1$	$c := [x]^{sc}$
a = b = 1, c = 0		

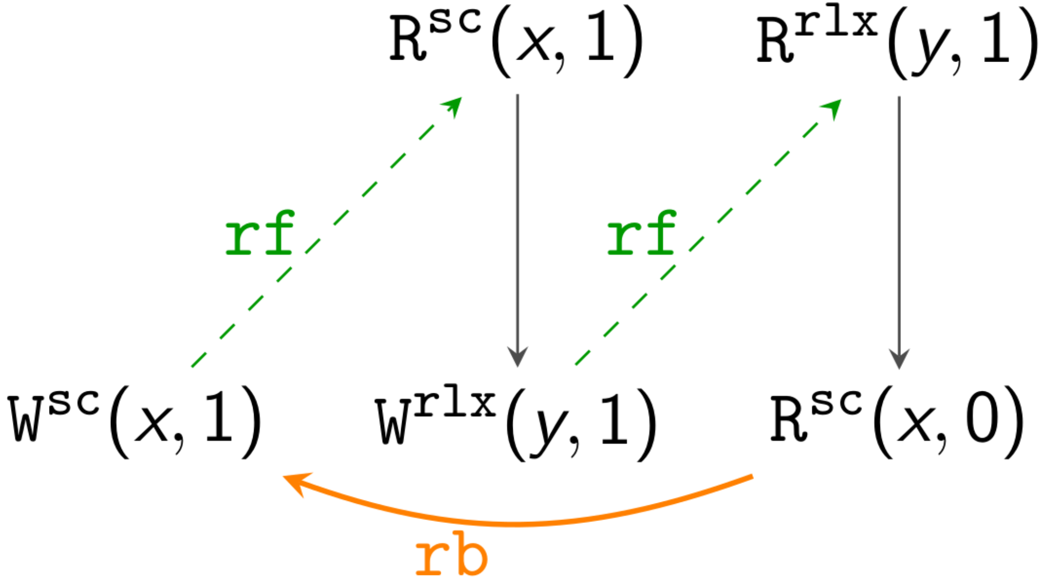


IMM: can have a cycle made of **po**, **rf** and **rb**

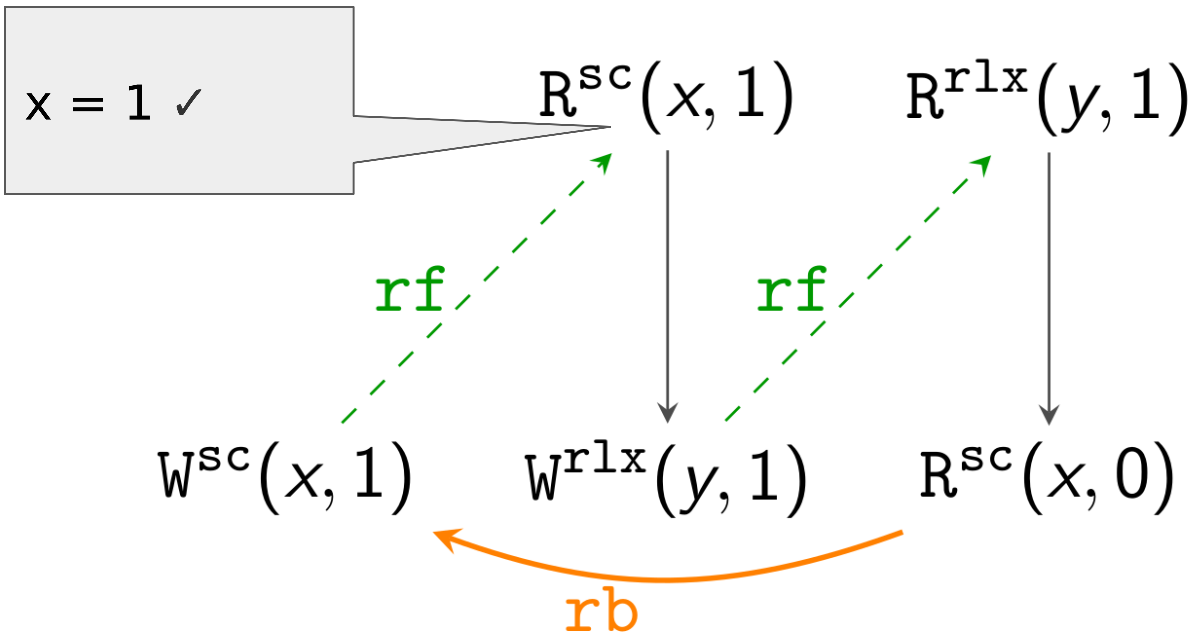
The identity compilation scheme won’t work



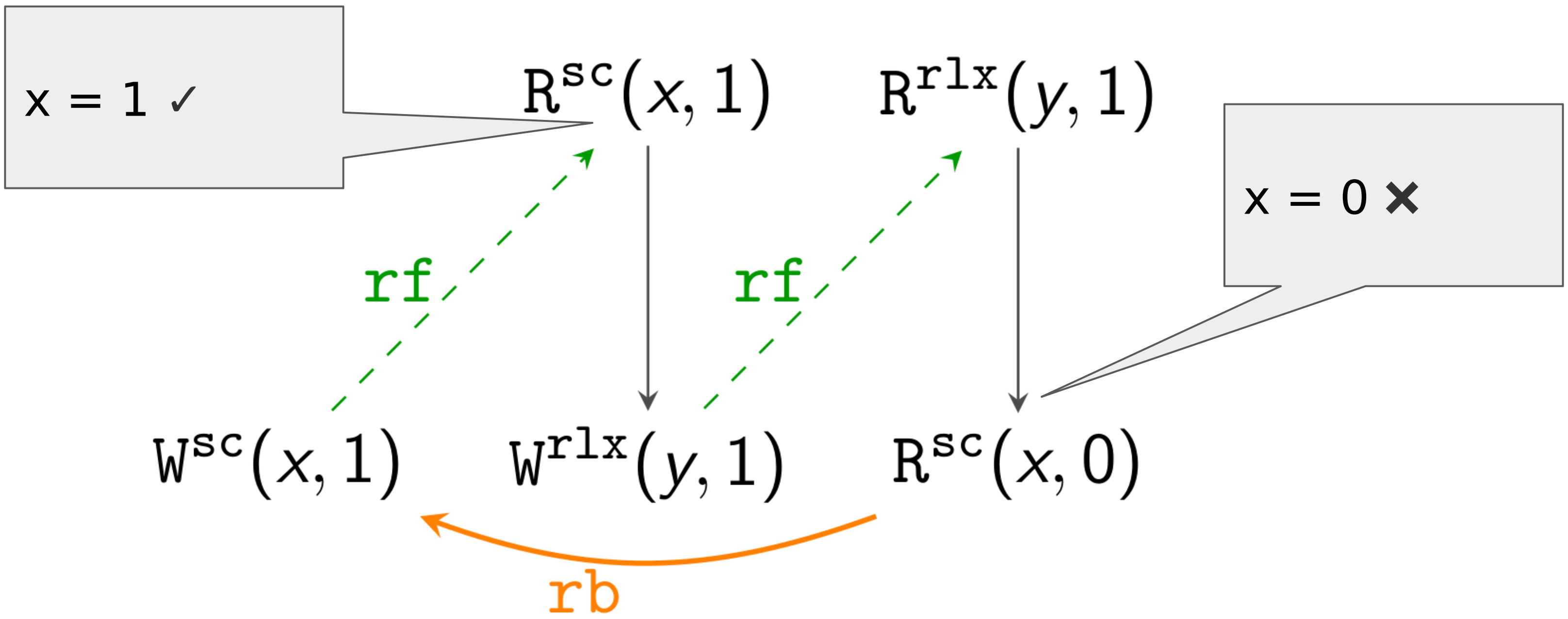
Observed writes should remain so



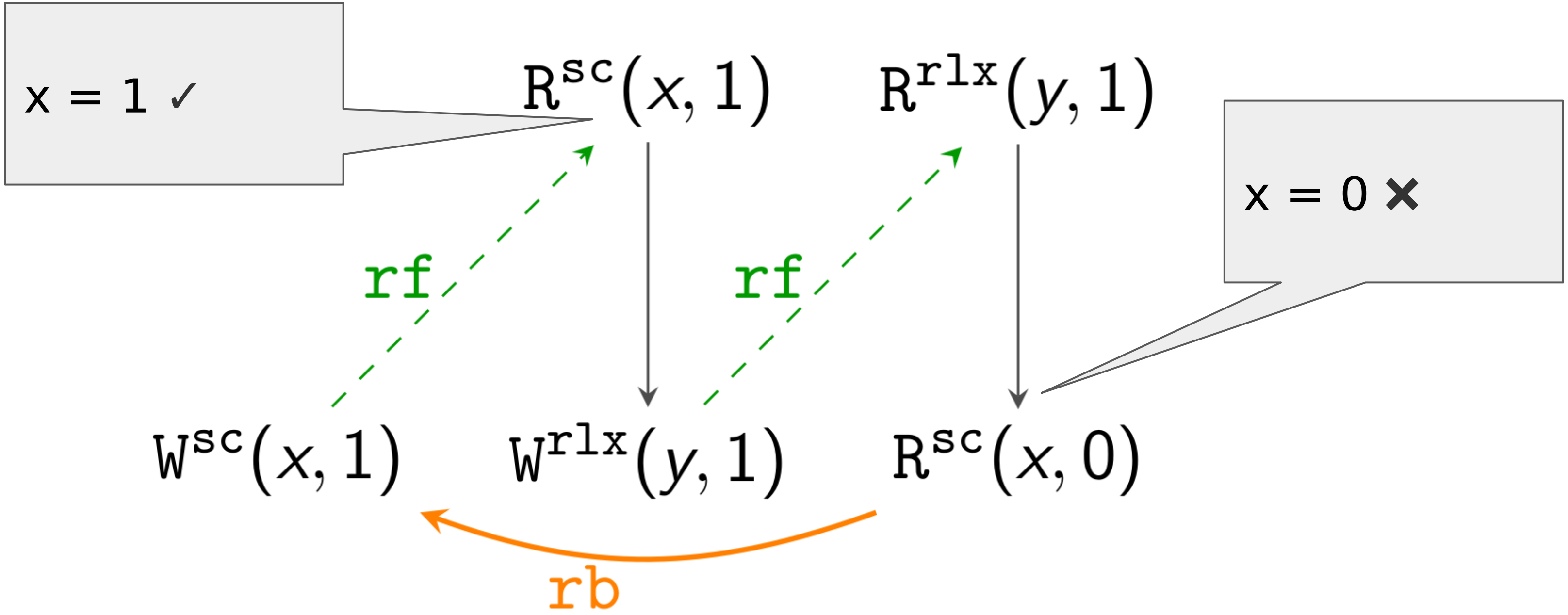
Observed writes should remain so



Observed writes should remain so



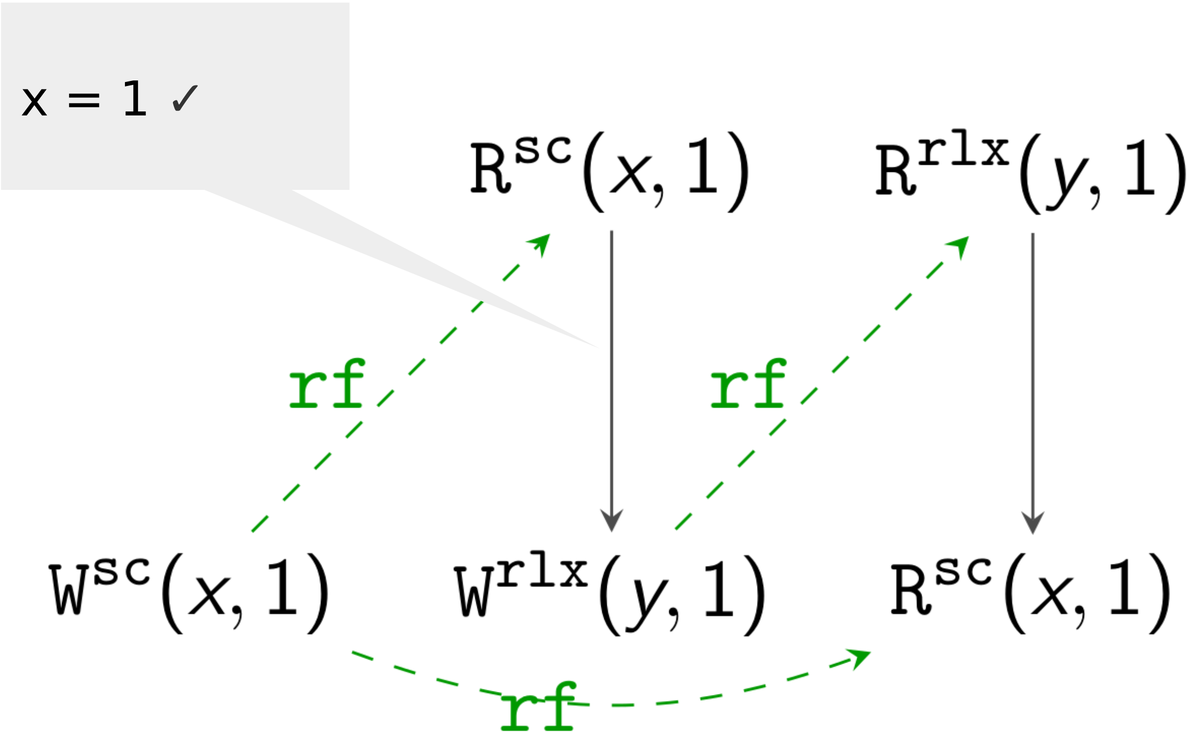
Observed writes should remain so
= graph should have no cycles with rb



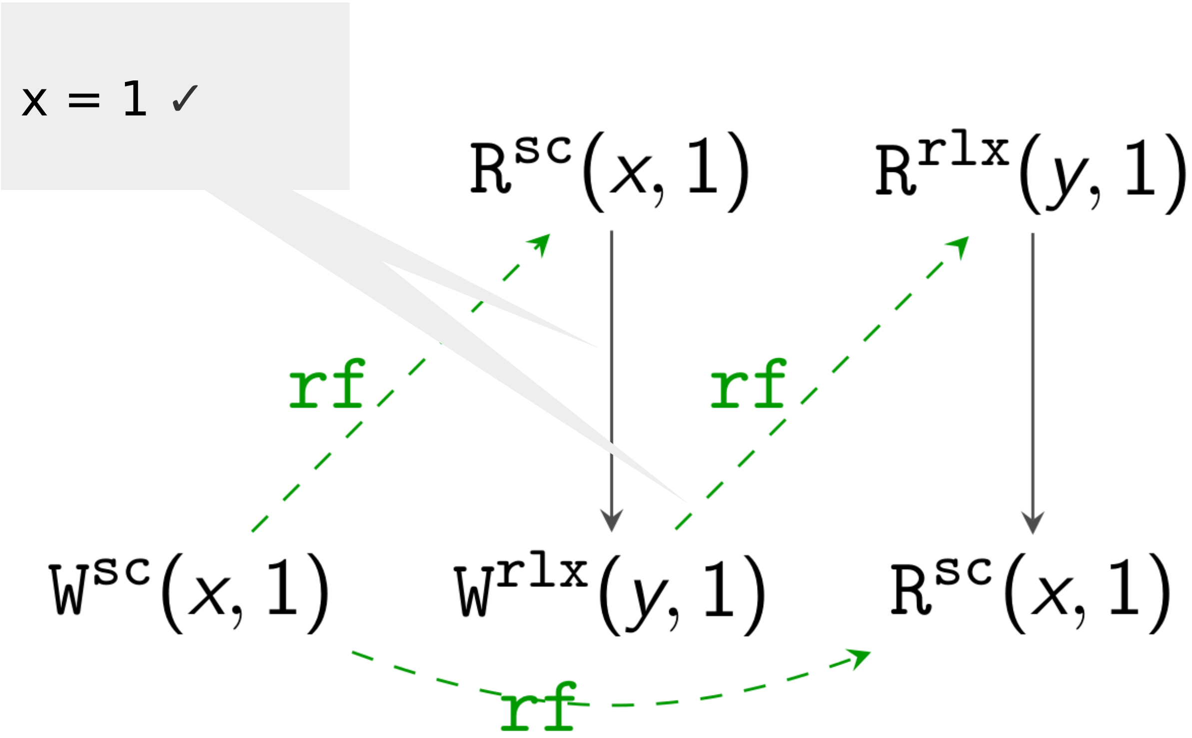
“Release” known writes and “acquire” them next

$$
\begin{tikzcd}
& \mathtt{R}^{\mathtt{sc}}(x, 1) \arrow[d] & \mathtt{R}^{\mathtt{rlx}}(y, 1) \arrow[d] \\
\mathtt{W}^{\mathtt{sc}}(x, 1) \arrow[ur, dashed, "\mathtt{rf}"] \arrow[rr, bend right, dashed, "\mathtt{rf}"'] & \mathtt{W}^{\mathtt{rlx}}(y, 1) \arrow[ur, dashed, "\mathtt{rf}"] & \mathtt{R}^{\mathtt{sc}}(x, 1)
\end{tikzcd}
$$

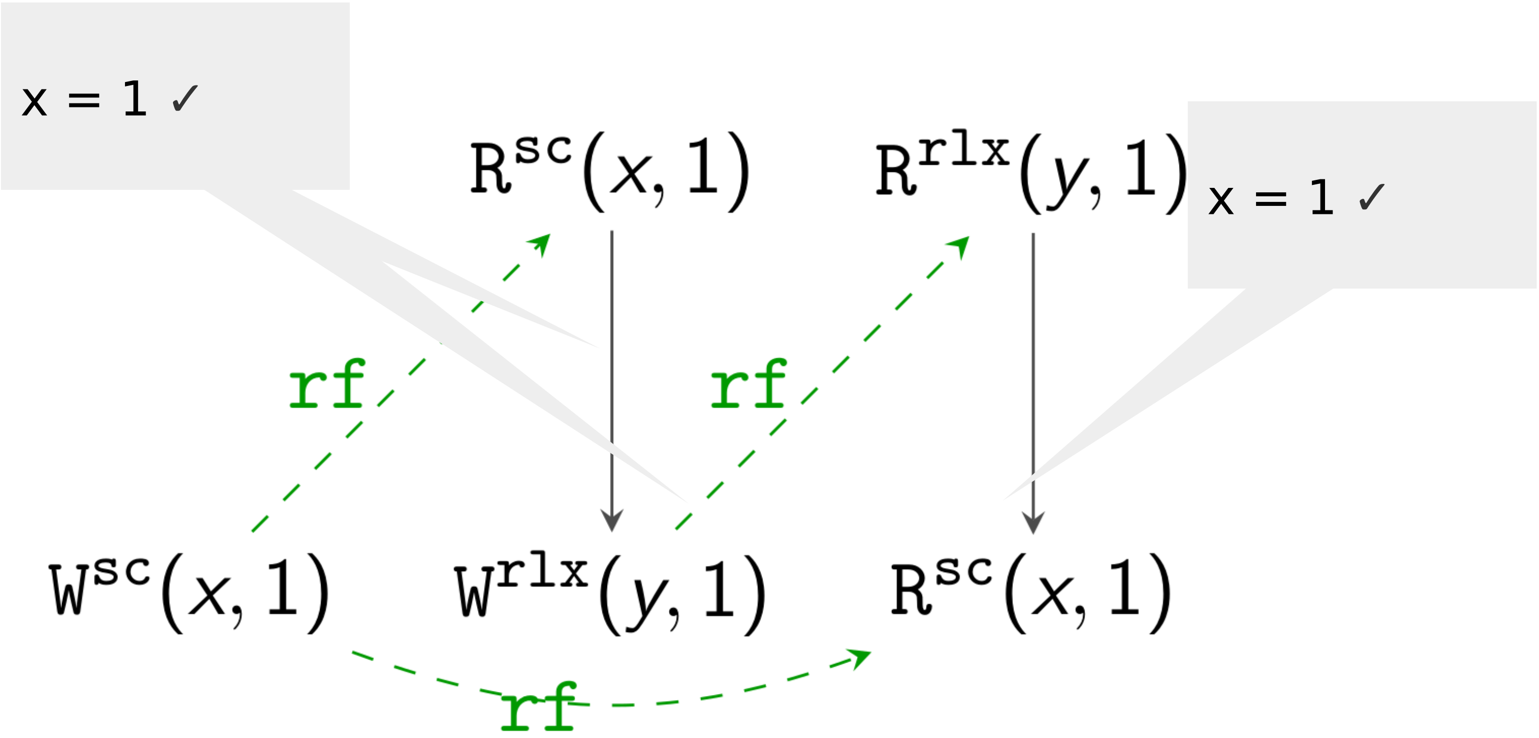
"Release" known writes and "acquire" them next



“Release” known writes and “acquire” them next



“Release” known writes and “acquire” them next



Implemented with release and acquire fences

Implemented with release and acquire fences

compile(Prog) = [na $\rightarrow$ rlx, at $\rightarrow$ sc]Prog + Fences^{rel} + Fences^{acq}

Implemented with release and acquire fences

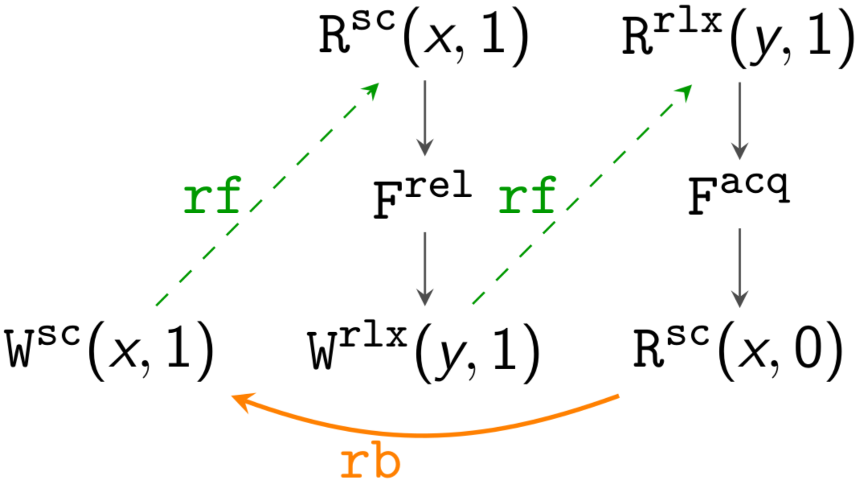
compile(Prog) = [na → rlx, at → sc]Prog + Fences^{rel} + Fences^{acq}

[x] = [y] = 0		
$[x]^{sc} := 1$	$a := [x]^{sc}$	$b := [y]^{rlx}$
	fence^{rel}	fence^{acq}
	$[y]^{rlx} := 1$	$c := [x]^{sc}$

Implemented with release and acquire fences

compile(Prog) = [na → rlx, at → sc]Prog + Fences^{rel} + Fences^{acq}

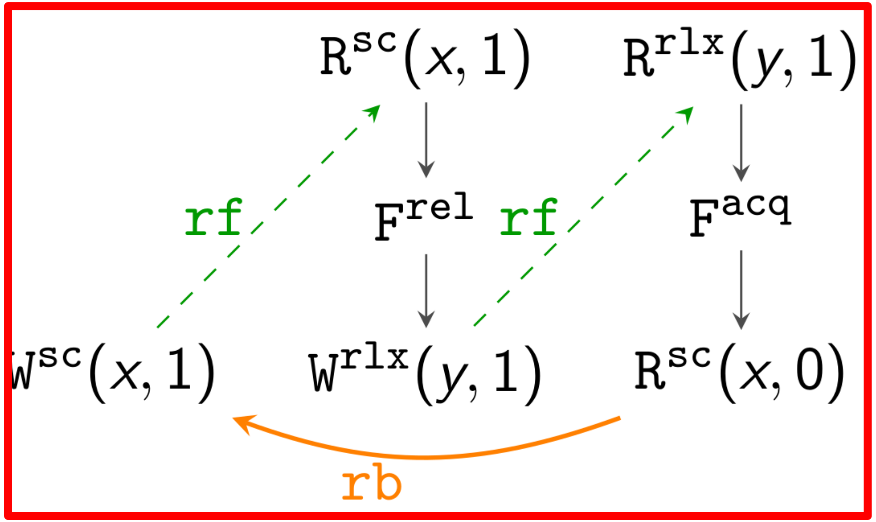
[x] = [y] = 0		
$[x]^{sc}$:= 1	a := $[x]^{sc}$	b := $[y]^{rlx}$
	fence^{rel}	fence^{acq}
	$[y]^{rlx}$:= 1	c := $[x]^{sc}$



Implemented with release and acquire fences

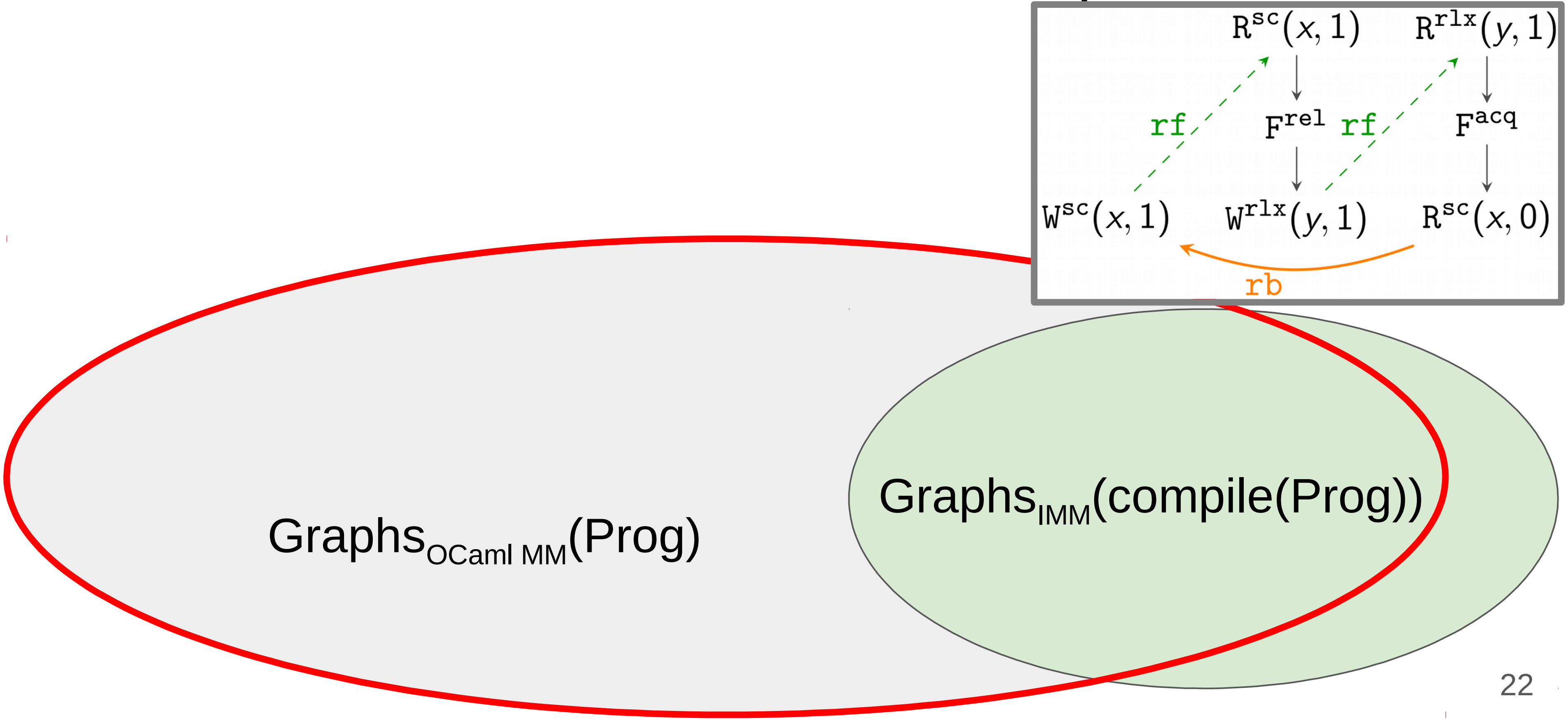
compile(Prog) = [na → rlx, at → sc]Prog + Fences^{rel} + Fences^{acq}

[x] = [y] = 0		
$[x]^{sc}$:= 1	a := $[x]^{sc}$	b := $[y]^{rlx}$
	fence^{rel}	fence^{acq}
	$[y]^{rlx}$:= 1	c := $[x]^{sc}$
~~a = b = 1, c = 0~~		



IMM: can have a cycle made of **po**, **rf** and **rb**
if there is **rf** without sc and fences

An IMM-inconsistent behavior is now prohibited



The resulting scheme prohibits unwanted behaviors

OCaml MM	IMM
$\mathtt{r} := [\mathtt{x}]^{\mathtt{na}}$	$\mathtt{r} := [\mathtt{x}]^{\mathtt{rlx}}$
$[\mathtt{x}]^{\mathtt{na}} := \mathtt{v}$	$\mathtt{fence}^{\mathtt{acqrel}}; [\mathtt{x}]^{\mathtt{rlx}} := \mathtt{v}$
$\mathtt{r} := [\mathtt{x}]^{\mathtt{at}}$	$\mathtt{fence}^{\mathtt{acq}}; \mathtt{r} := [\mathtt{x}]^{\mathtt{sc}}$
$[\mathtt{x}]^{\mathtt{at}} := \mathtt{v}$	$\mathtt{fence}^{\mathtt{acq}}; \mathtt{exchange}^{\mathtt{sc}}(\mathtt{x}, \mathtt{v})$

Takeaway

- Compilation scheme from OCaml MM to IMM
- Proved to be correct
- Formalized in Coq

Machine-verified compilation scheme from OCaml MM to Power

https://github.com/weakmemory/imm