Overhauling SC atomics in C11 and OpenCL

John Wickerson, Mark Batty, and Alastair F. Donaldson

Imperial Concurrency Workshop
July 2015

TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are

TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are

  😕 too **complex**,

TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are

  😕 too **complex**,

  😟 too **weak**, and

TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are

  😕 too **complex**,

  😟 too **weak**, and

  😠 too **strong**.

TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are

  😕 too **complex**,

  😟 too **weak**, and

  😠 too **strong**.

- We suggest how to fix them 😌.

Outline

- Introduction to the C11 memory model
- Overhauling the rules for SC atomics in C11
- Introduction to the OpenCL memory model
- Overhauling the rules for SC atomics in OpenCL

C11 atomics

- A collection of indivisible operations for lock-free programming, e.g.:

```
atomic_store_explicit(x, 1, memory_order_relaxed);
```

C11 atomics

- A collection of indivisible operations for lock-free programming, e.g.:

```
atomic_store_explicit(x, 1, memory_order_release);
                            memory_order_relaxed
```

C11 atomics

- A collection of indivisible operations for lock-free programming, e.g.:

```
atomic_store_explicit(x, 1, memory_order_acquire);
                            memory_order_release
                            memory_order_relaxed
```

C11 atomics

- A collection of indivisible operations for lock-free programming, e.g.:

```
atomic_store_explicit(x, 1, memory_order_rel_acq);
                            memory_order_acquire
                            memory_order_release
                            memory_order_relaxed
```

C11 atomics

- A collection of indivisible operations for lock-free programming, e.g.:

```
atomic_store_explicit(x, 1, memory_order_seq_cst);
                            memory_order_rel_acq
                            memory_order_acquire
                            memory_order_release
                            memory_order_relaxed
```

C11 atomics

- A collection of indivisible operations for lock-free programming, e.g.:

```
atomic_store_explicit(x, 1, memory_order_seq_cst);
                            memory_order_rel_acq
                            memory_order_acquire
                            memory_order_release
                            memory_order_relaxed
```

The presence of these other memory orders makes the semantics of SC atomics surprisingly complex

C11 memory model

- Trace-based semantics ("executions").

C11 memory model

- Trace-based semantics ("executions").
- First phase: generate an overapproximation, by considering each thread in isolation.

C11 memory model

- Trace-based semantics ("executions").
- First phase: generate an overapproximation, by considering each thread in isolation.
- Second phase: remove executions that are inconsistent with the axioms of the memory model.

Example

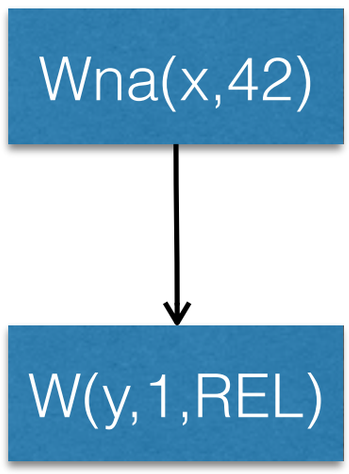
```
*x = 42;                              ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,           ||      memory_order_acquire))
  memory_order_release);              ||    print(*x);
```

Example

```
*x = 42;                              ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,           ||      memory_order_acquire))
  memory_order_release);              ||    print(*x);
```

Example

```
*x = 42;                                  ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,               ||      memory_order_acquire))
  memory_order_release);                  ||    print(*x);
```



Example

```
*x = 42;                          ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,       ||      memory_order_acquire))
  memory_order_release);          ||    print(*x);
```

Example

```
*x = 42;
atomic_store_explicit(y, 1,
  memory_order_release);
```

```
if (atomic_load_explicit(y,
    memory_order_acquire))
  print(*x);
```

Example

```
*x = 42;
atomic_store_explicit(y, 1,
  memory_order_release);
```

```
if (atomic_load_explicit(y,
    memory_order_acquire))
  print(*x);
```

Example

```
*x = 42;                                  ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,               ||      memory_order_acquire))
  memory_order_release);                  ||    print(*x);
```

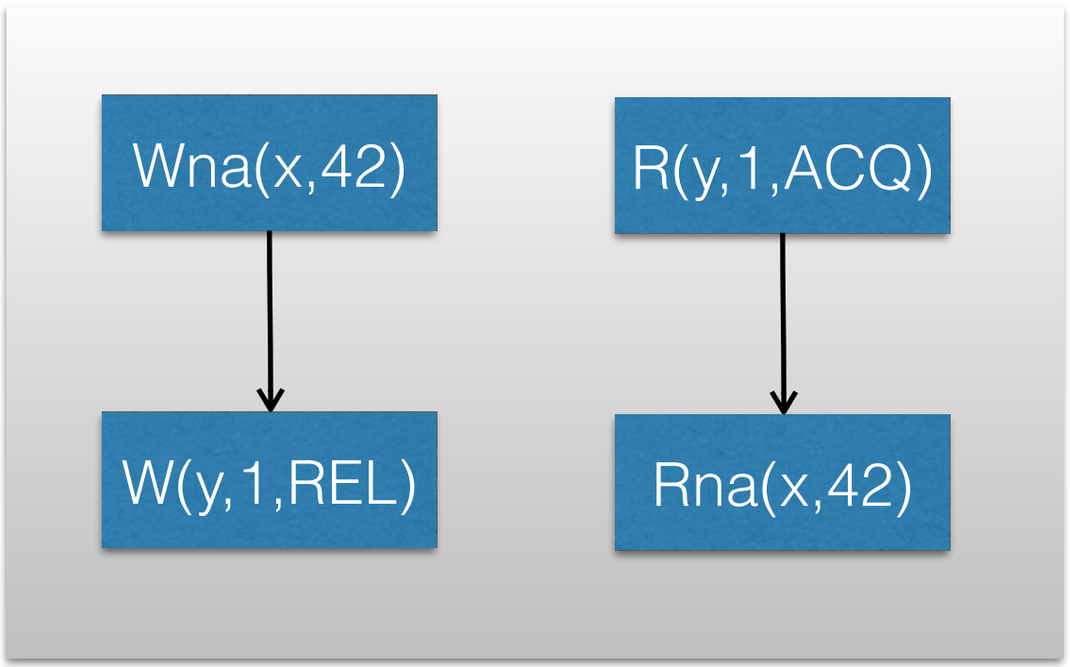
Example

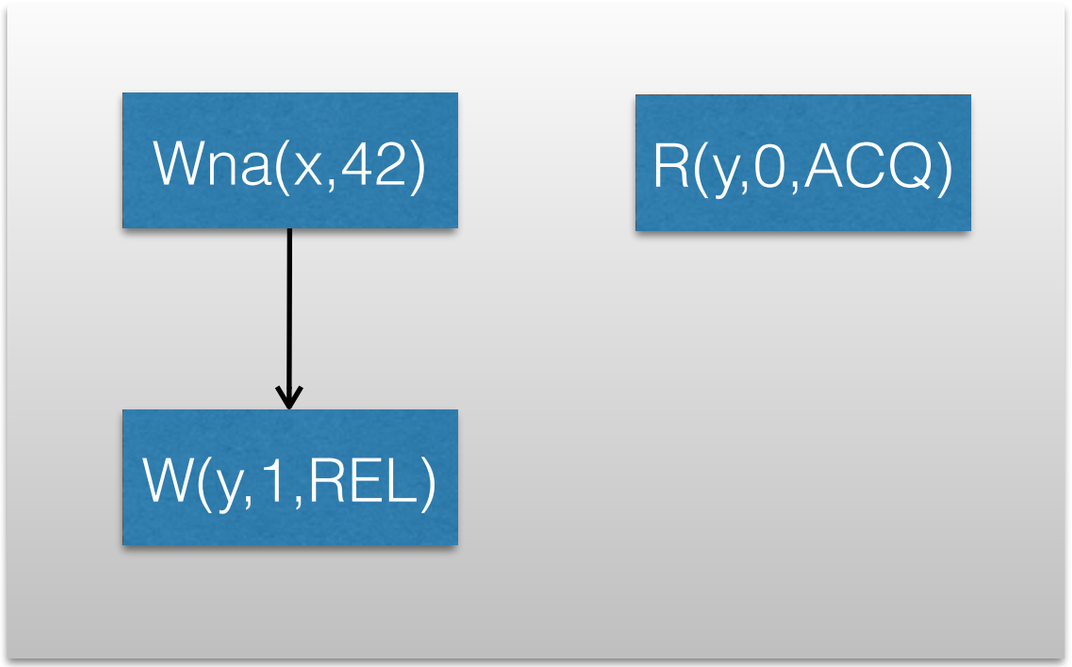
```
*x = 42;                                  ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,               ||      memory_order_acquire))
  memory_order_release);                  ||    print(*x);
```

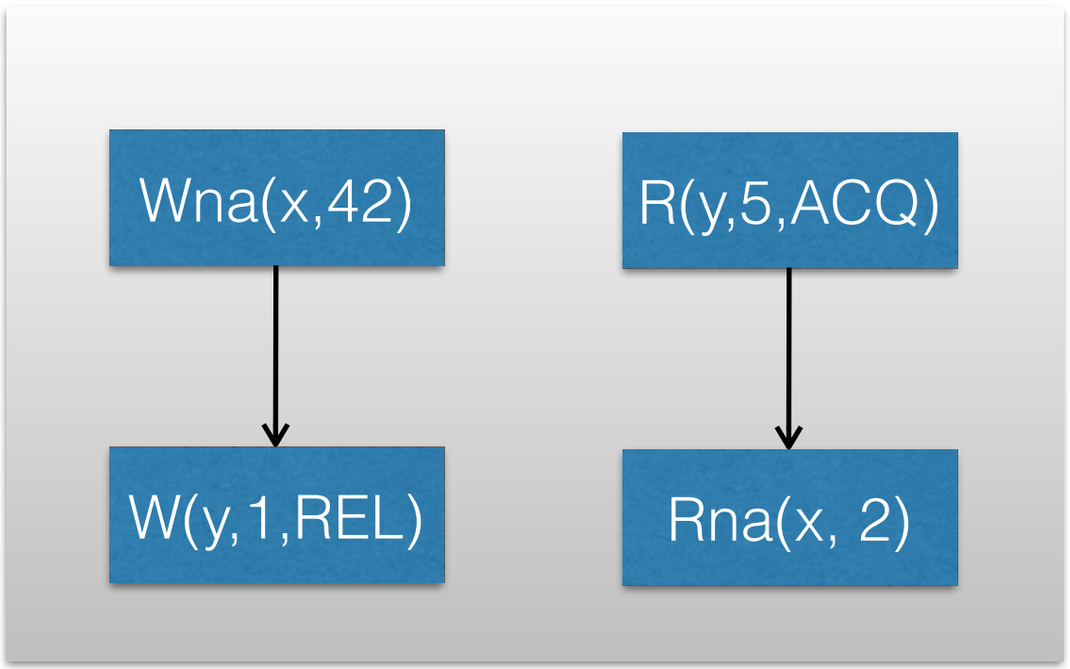
Example

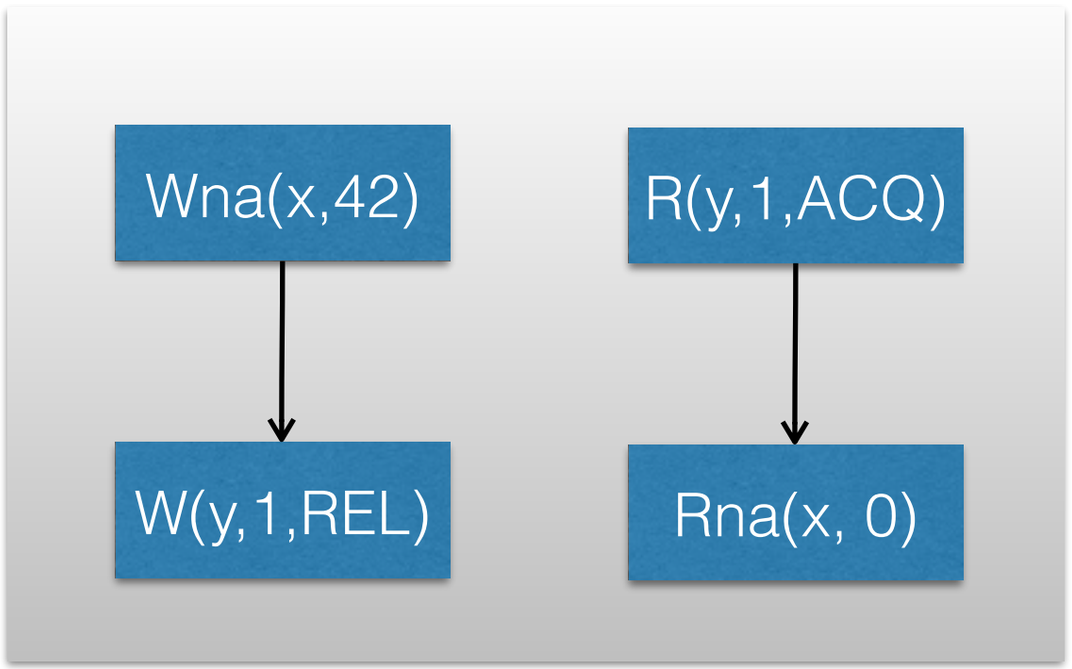
```
*x = 42;                                  ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,               ||      memory_order_acquire))
  memory_order_release);                  ||    print(*x);
```

Example

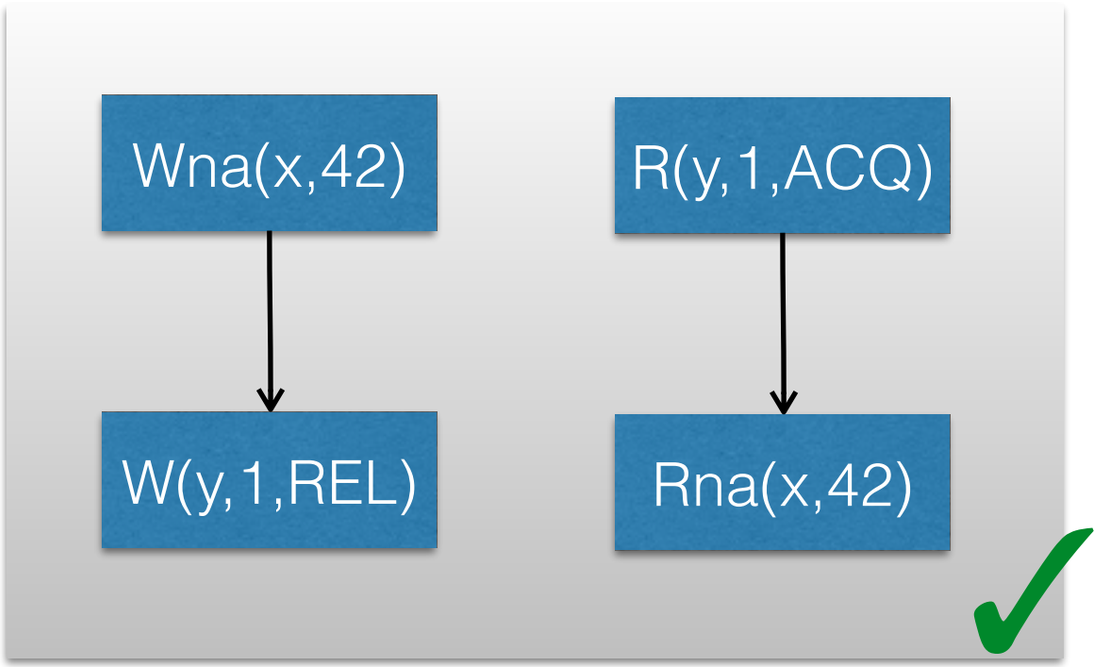

Wna(x,42)
W(y,1,REL)
R(y,1,ACQ)
Rna(x,42)

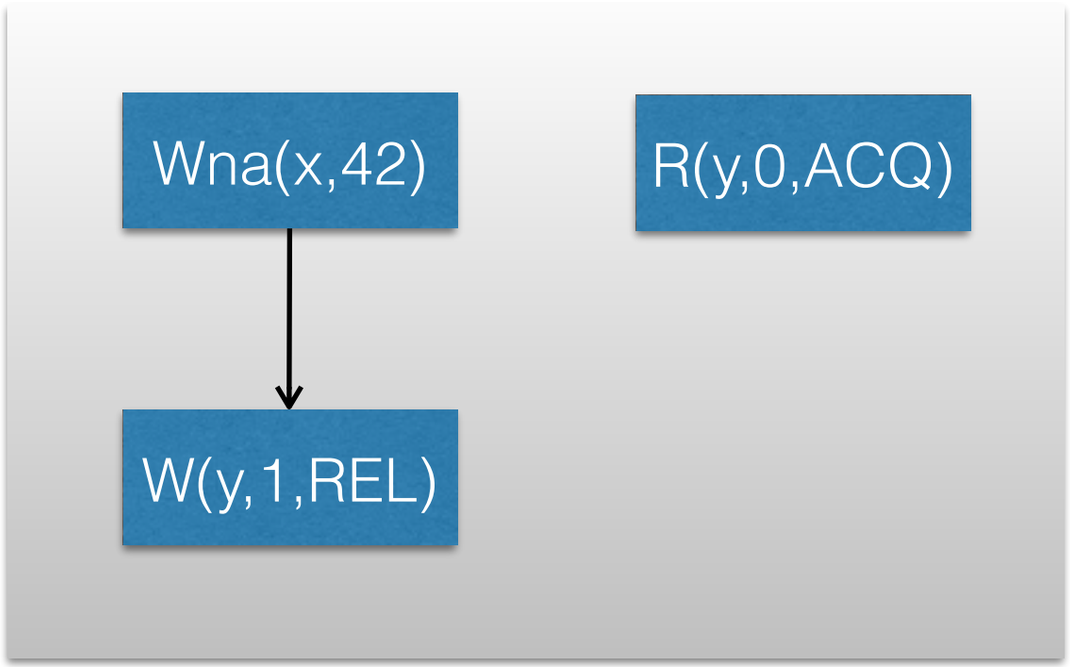

Wna(x,42)
W(y,1,REL)
R(y,0,ACQ)

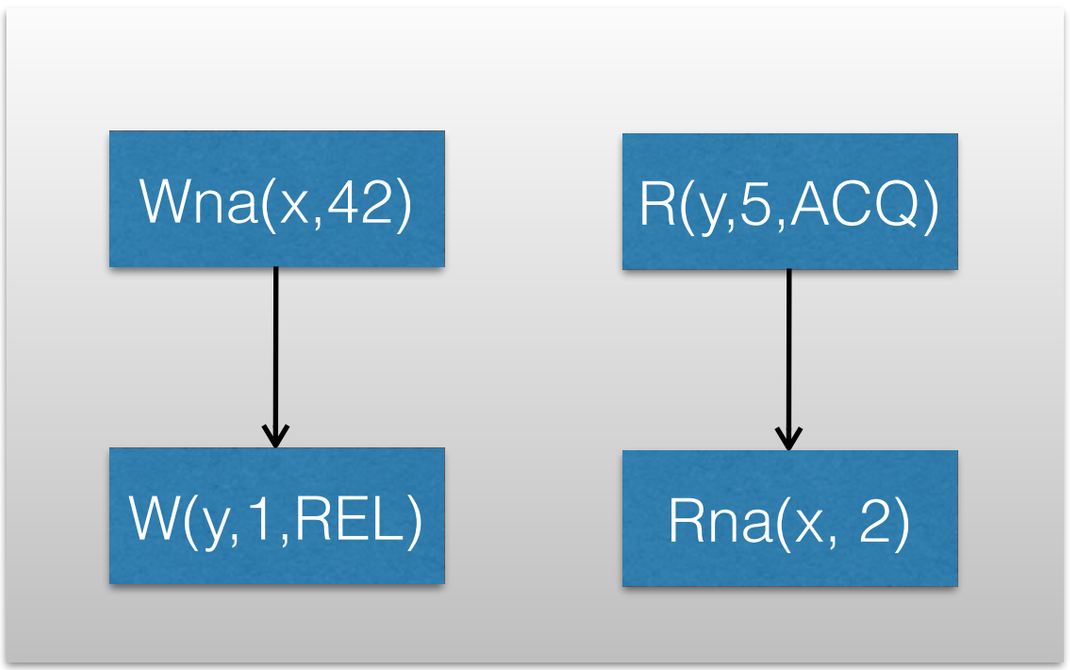

Wna(x,42)
W(y,1,REL)
R(y,5,ACQ)
Rna(x, 2)

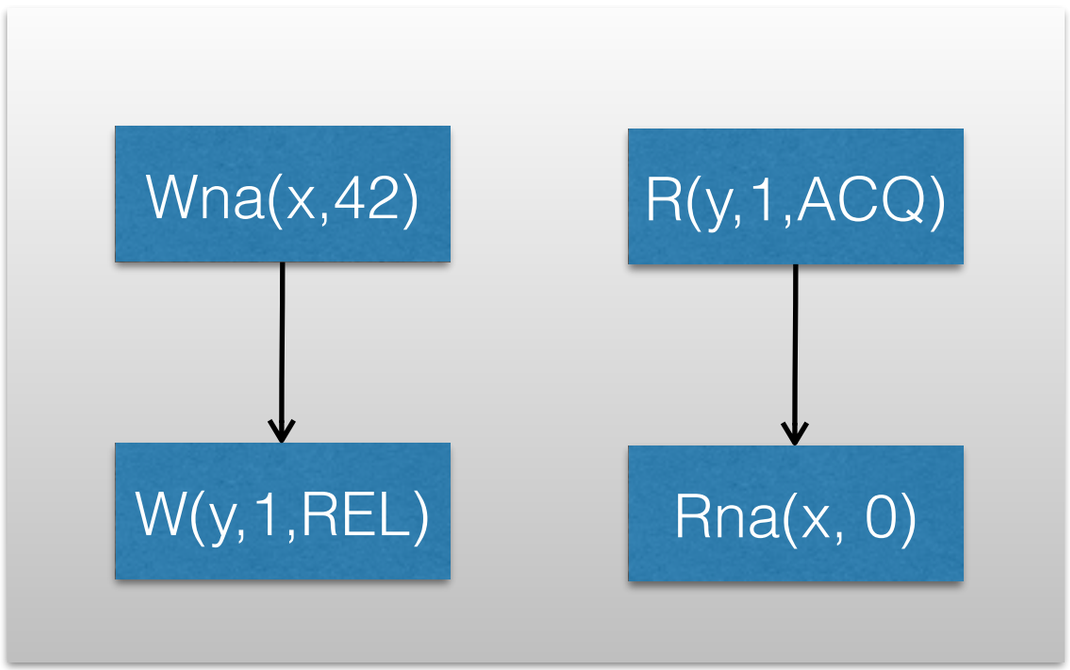

Wna(x,42)
W(y,1,REL)
R(y,1,ACQ)
Rna(x, 0)

Example

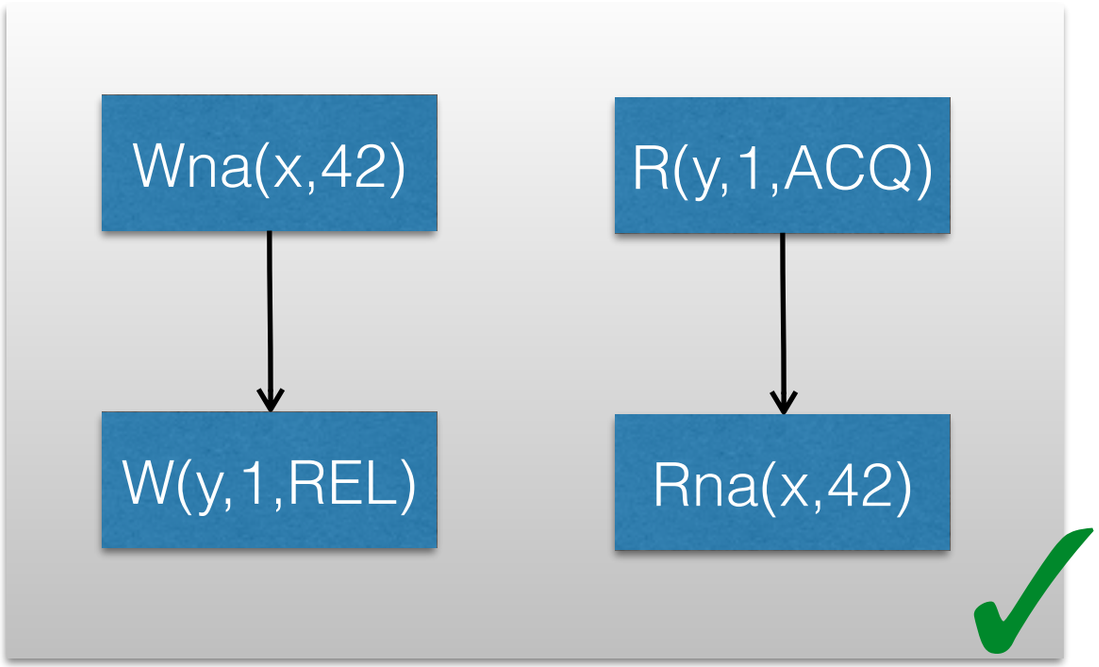

Wna(x,42)
W(y,1,REL)
R(y,1,ACQ)
Rna(x,42)

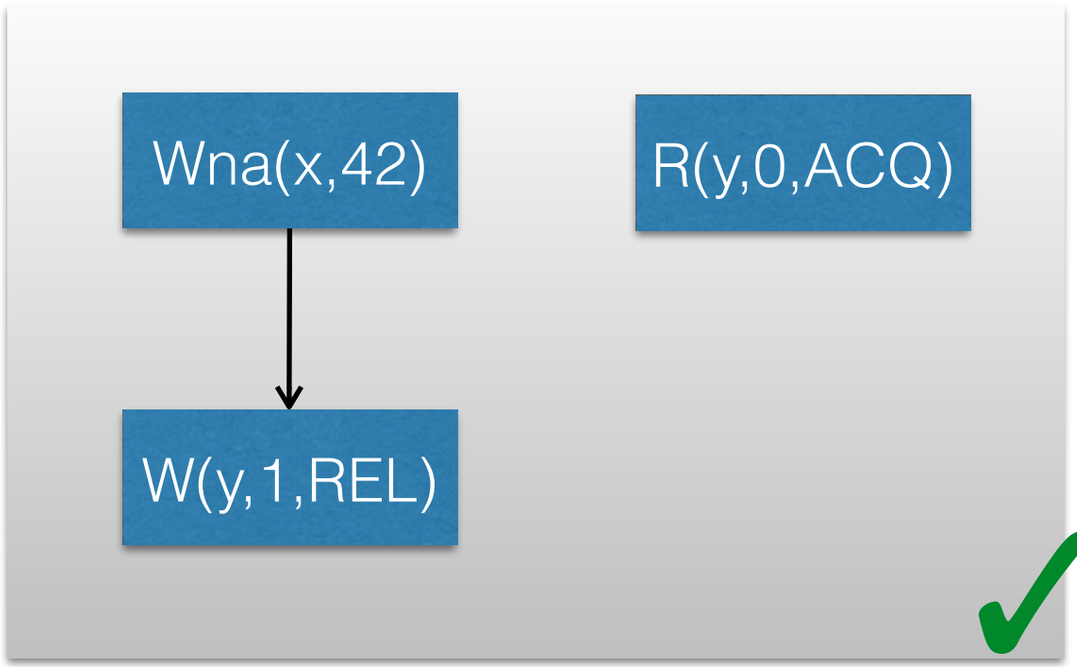

Wna(x,42)
W(y,1,REL)
R(y,0,ACQ)

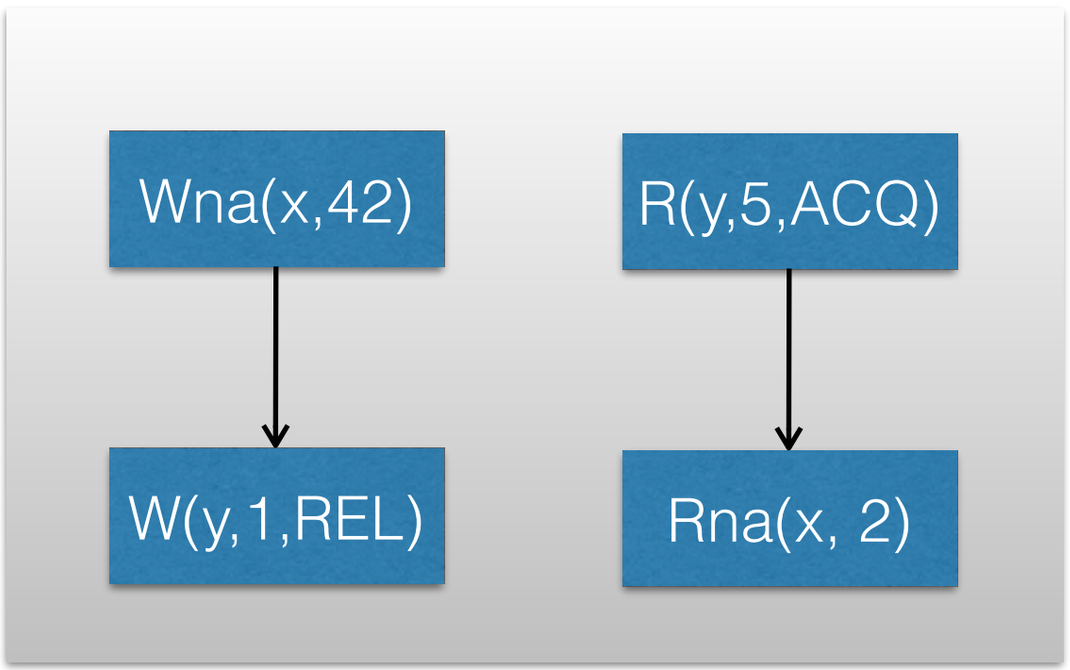

Wna(x,42)
W(y,1,REL)
R(y,5,ACQ)
Rna(x, 2)

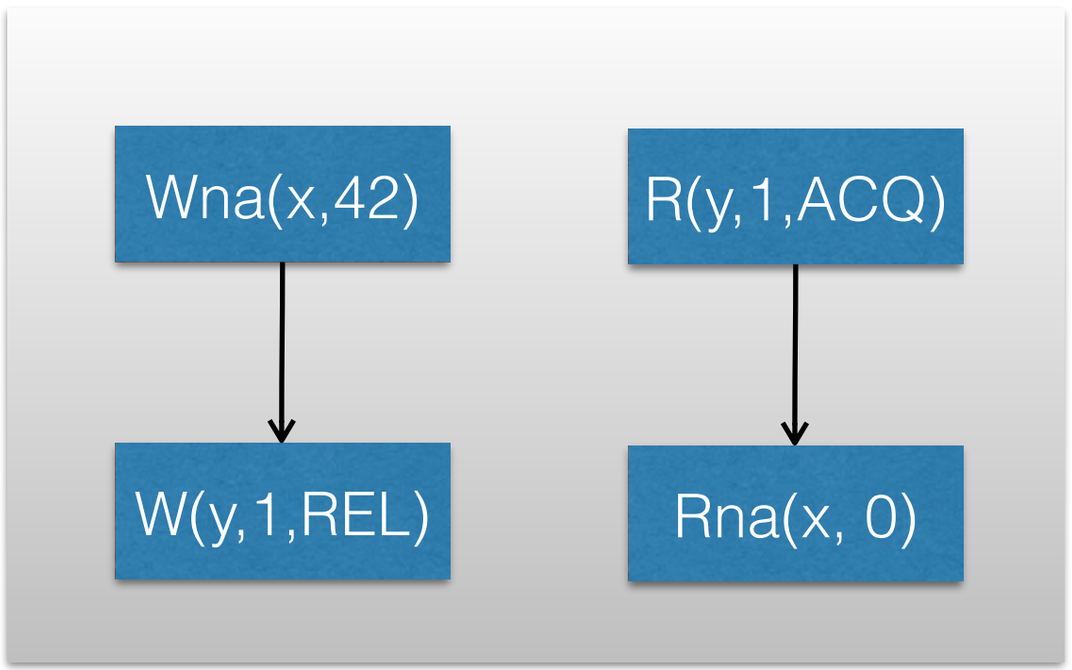

Wna(x,42)
W(y,1,REL)
R(y,1,ACQ)
Rna(x, 0)

Example

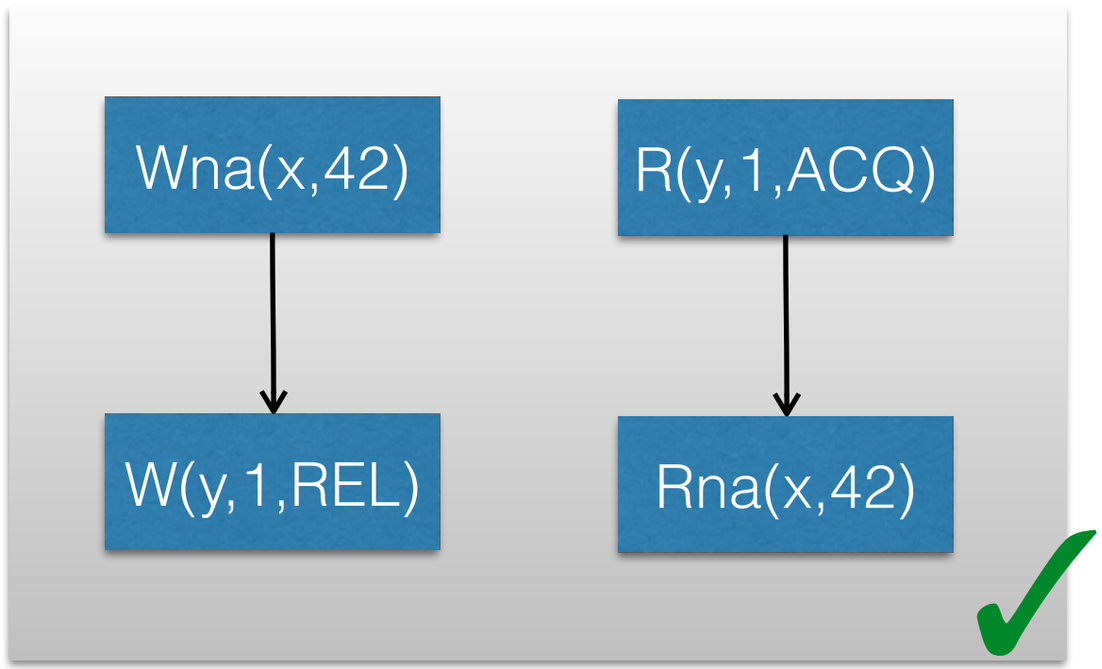

Wna(x,42)
W(y,1,REL)
R(y,1,ACQ)
Rna(x,42)

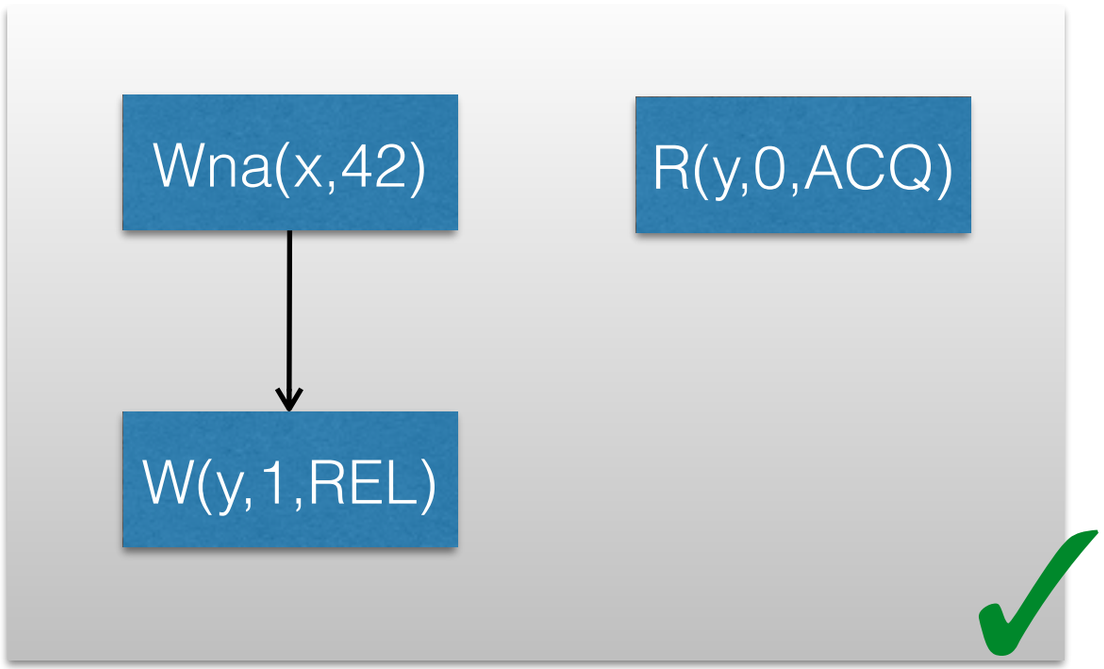

Wna(x,42)
W(y,1,REL)
R(y,0,ACQ)

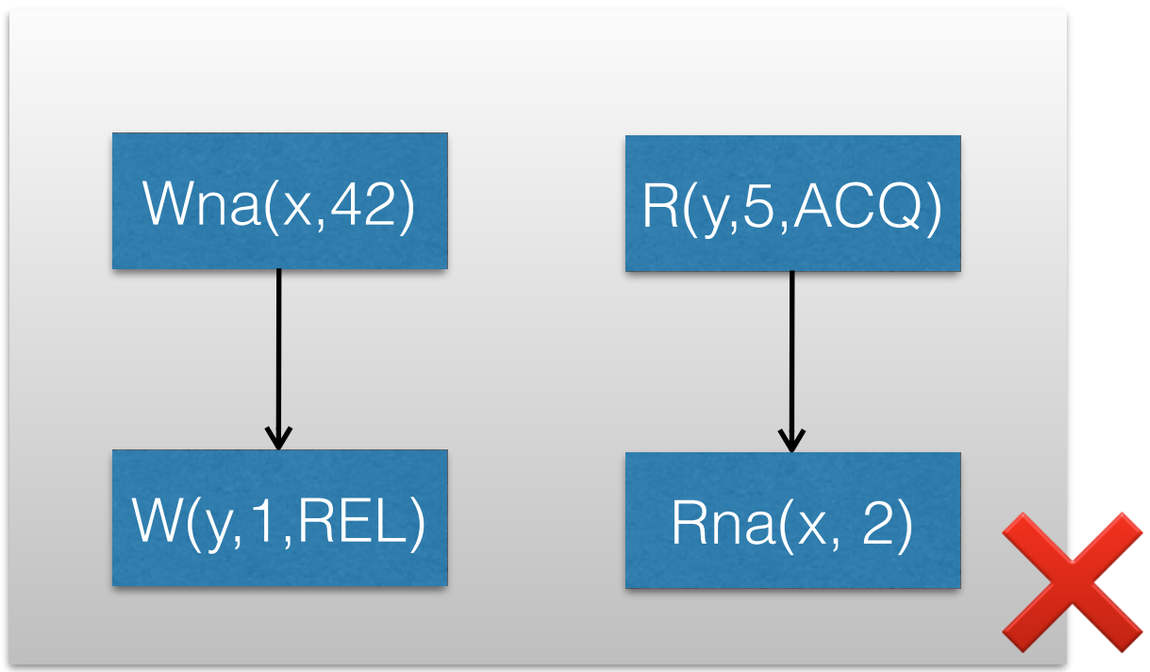

Wna(x,42)
W(y,1,REL)
R(y,5,ACQ)
Rna(x, 2)

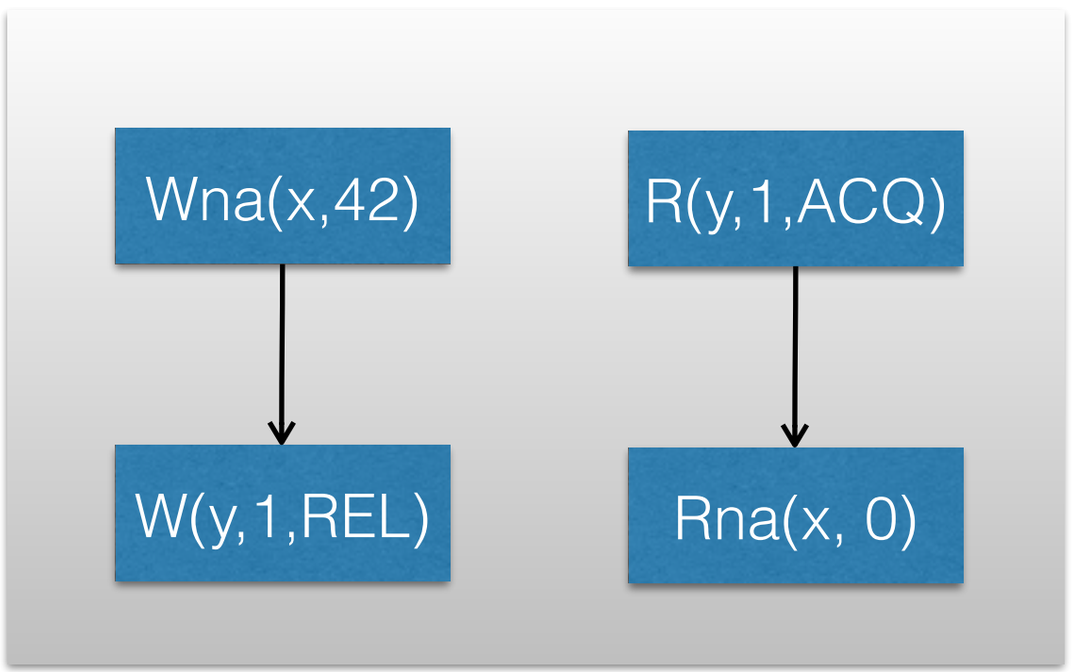

Wna(x,42)
W(y,1,REL)
R(y,1,ACQ)
Rna(x, 0)

Example

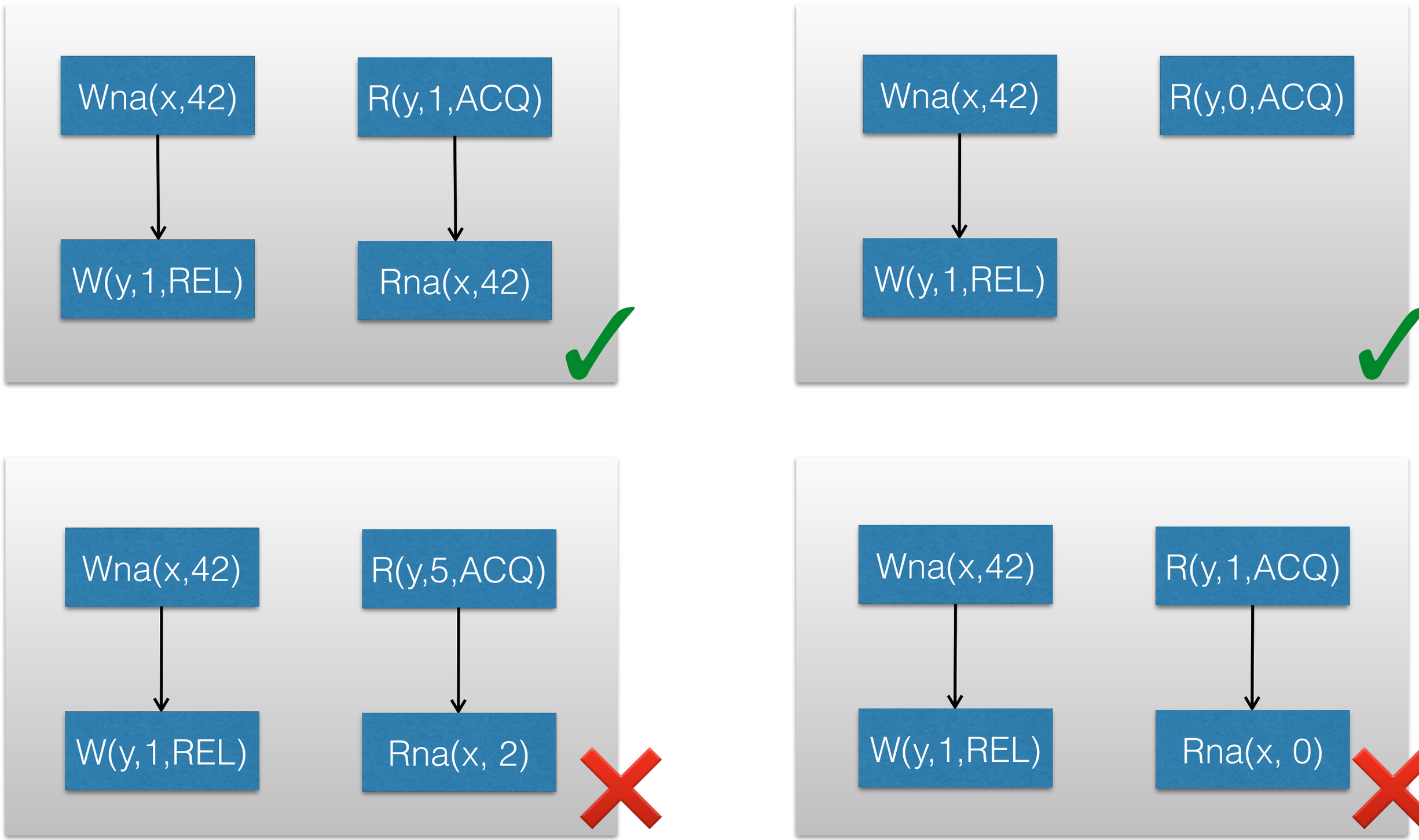

Wna(x,42)
R(y,1,ACQ)
W(y,1,REL)
Rna(x,42)


Wna(x,42)
R(y,0,ACQ)
W(y,1,REL)


Wna(x,42)
R(y,5,ACQ)
W(y,1,REL)
Rna(x, 2)


Wna(x,42)
R(y,1,ACQ)
W(y,1,REL)
Rna(x, 0)

Example


Wna(x,42)
W(y,1,REL)
R(y,1,ACQ)
Rna(x,42)
Wna(x,42)
W(y,1,REL)
R(y,0,ACQ)
Wna(x,42)
W(y,1,REL)
R(y,5,ACQ)
Rna(x, 2)
Wna(x,42)
W(y,1,REL)
R(y,1,ACQ)
Rna(x, 0)

Consistent executions

Consistent executions

- Execution *X* is **consistent** iff

satisfies all the **consistency axioms**.

Consistent executions

- Execution *X* is **consistent** iff

satisfies all the **consistency axioms**.

- ⟦P⟧ = P's consistent executions*

Consistent executions

- Execution *X* is **consistent** iff

satisfies all the **consistency axioms**.

- ⟦P⟧ = P's consistent executions*

*unless P also admits a **faulty** execution, then ⟦P⟧ = any execution

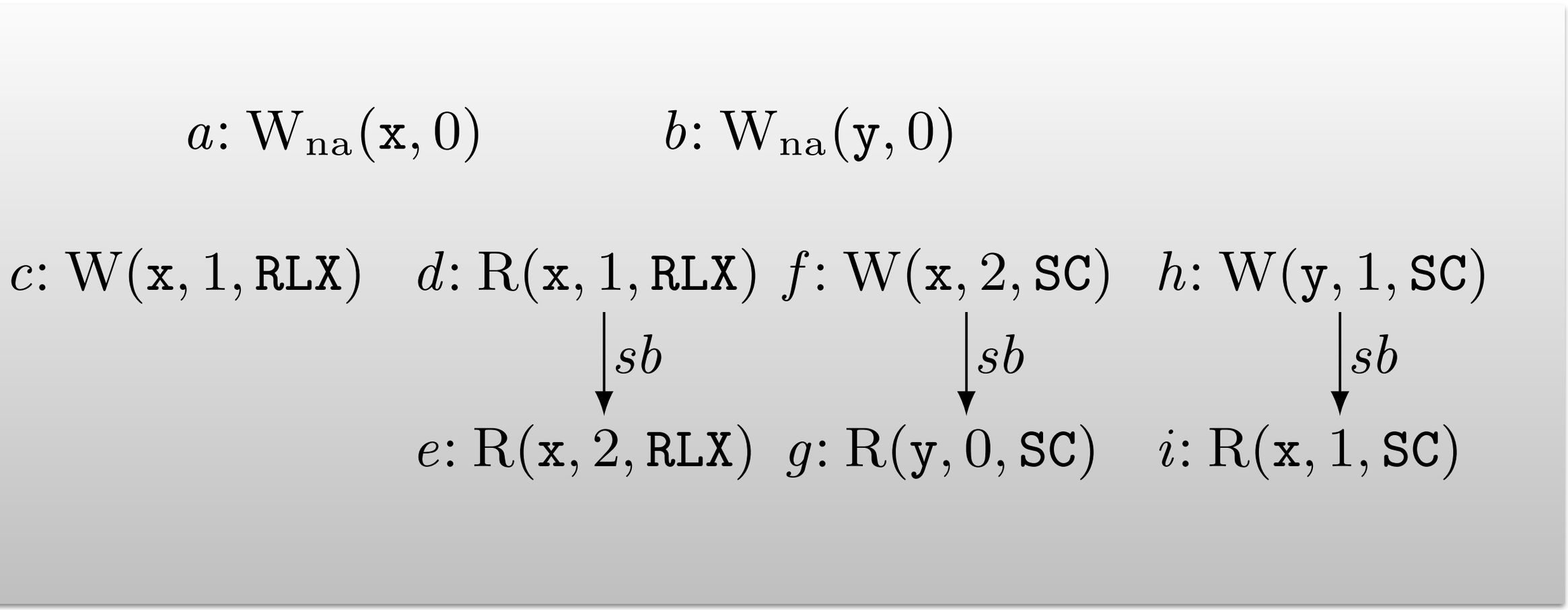
Consistent executions

- Execution *X* is **consistent** iff
  there exists *rf*, *mo* and *S* such that
    (*X*,*rf*,*mo*,*S*) is well-formed and
      satisfies all the **consistency axioms**.

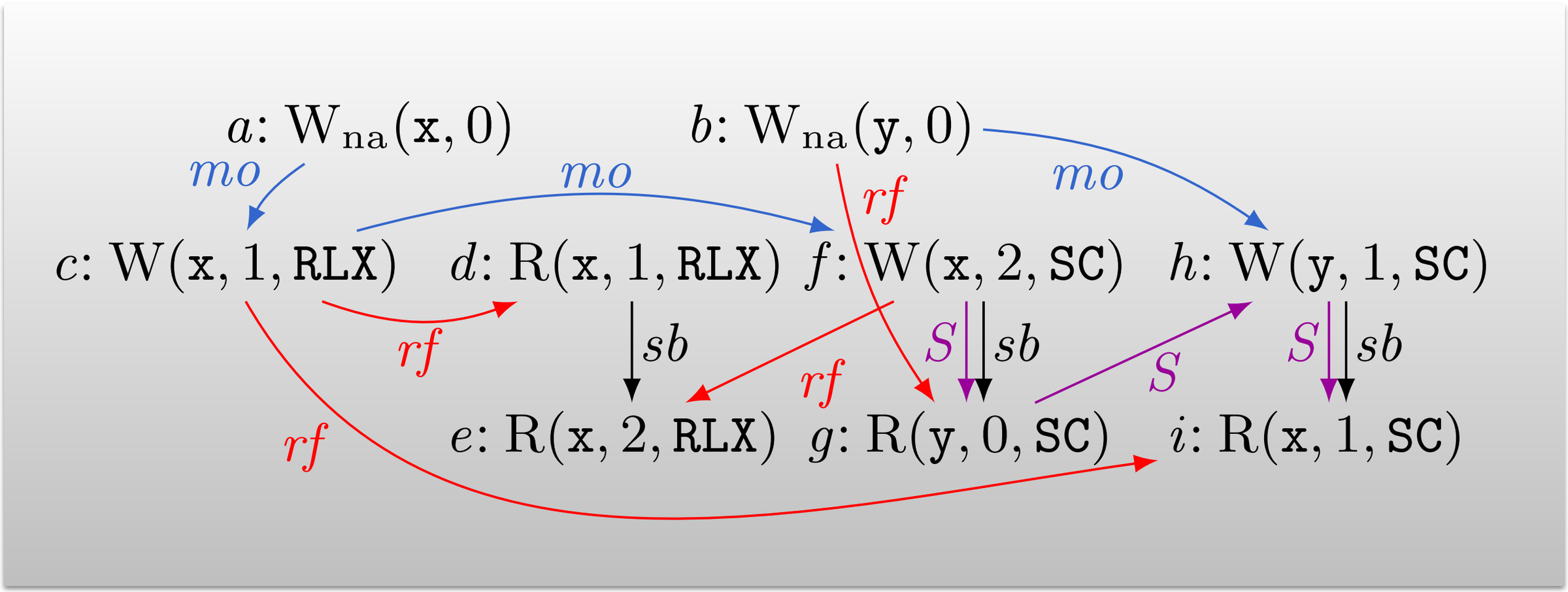
- ⟦P⟧ = P's consistent executions*

*unless P also admits a **faulty** execution, then ⟦P⟧ = any execution

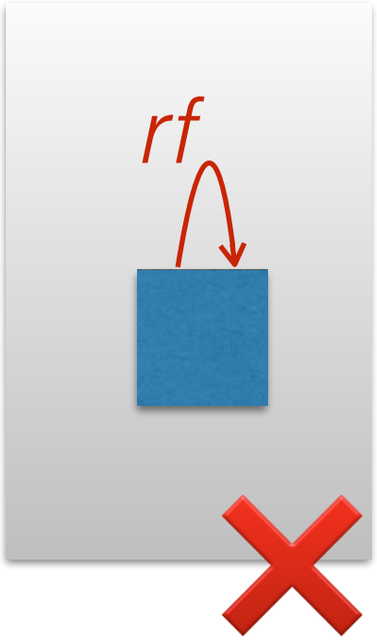
Candidate executions

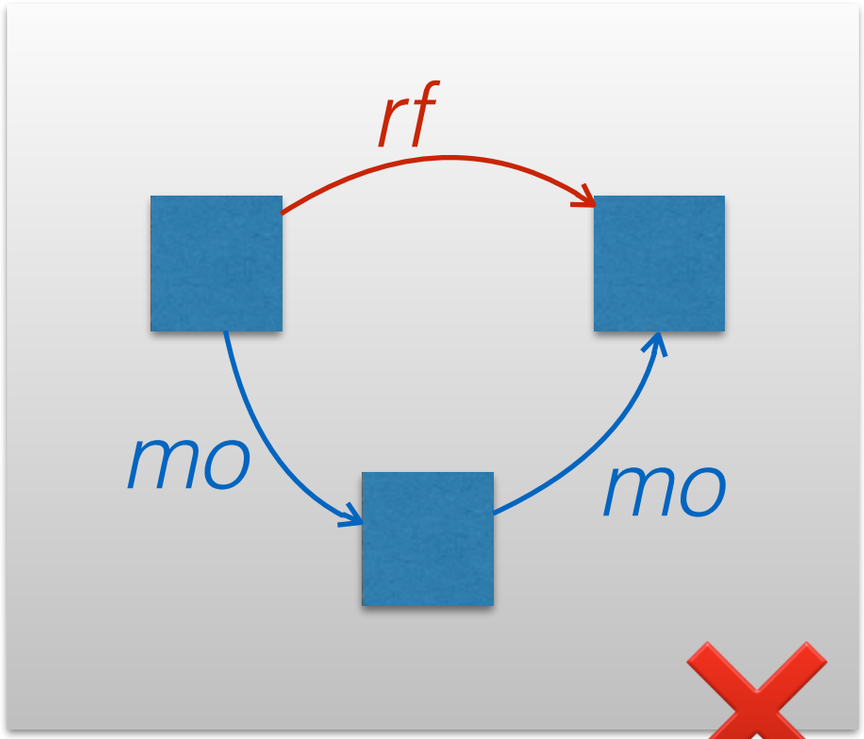


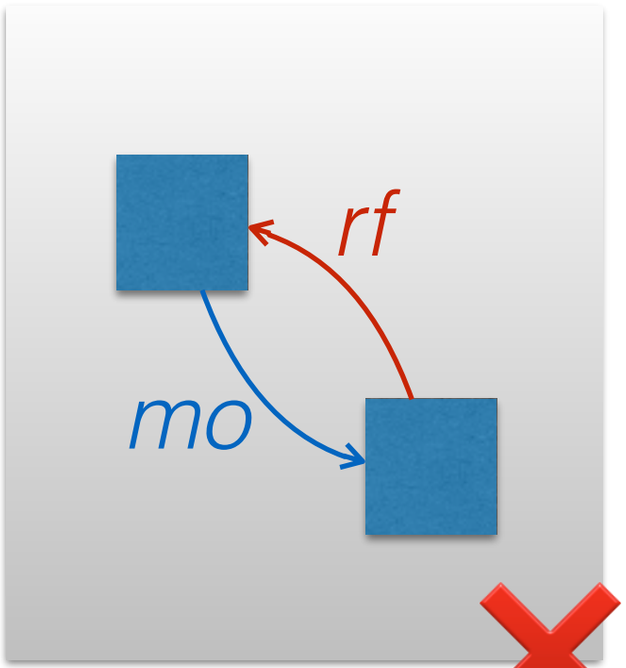
Candidate executions



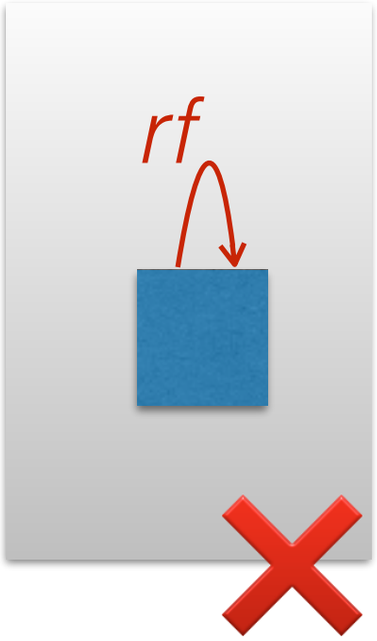
Some axioms

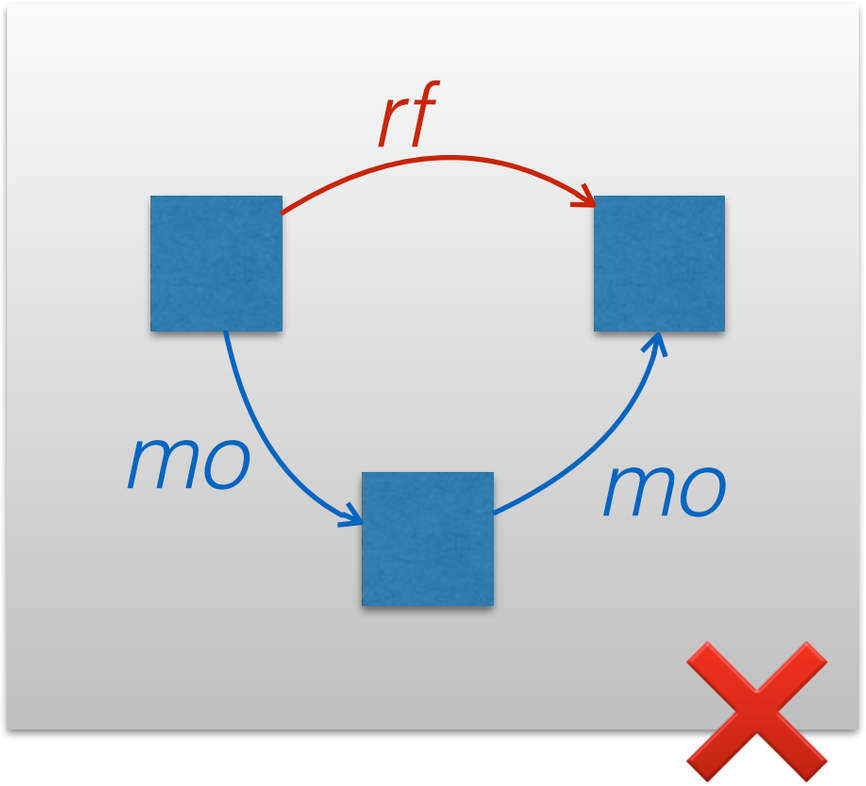


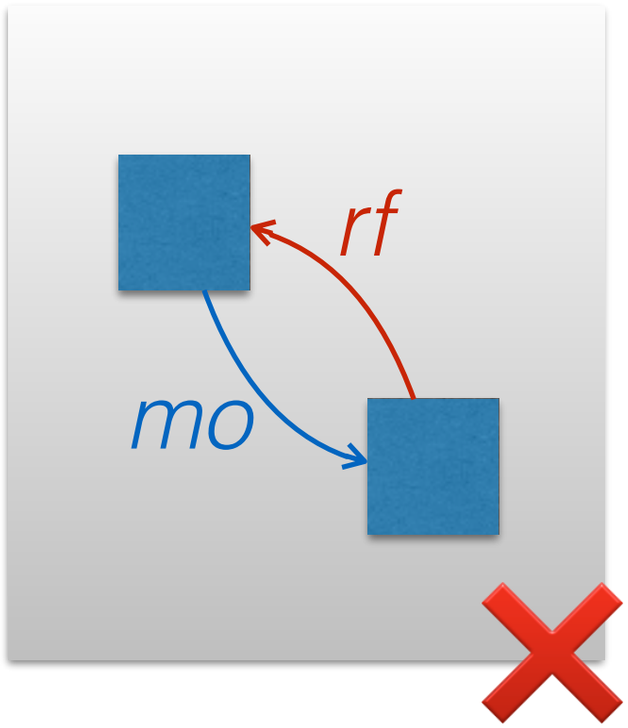




Some axioms

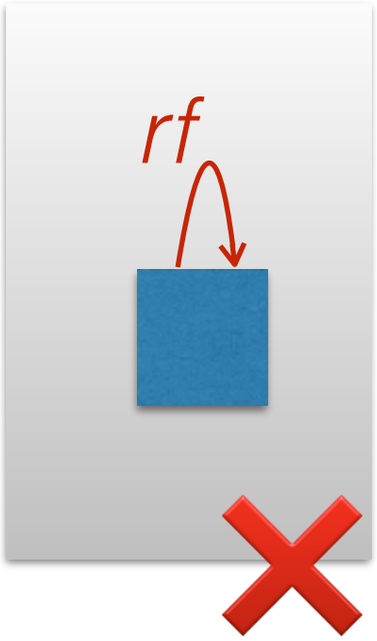


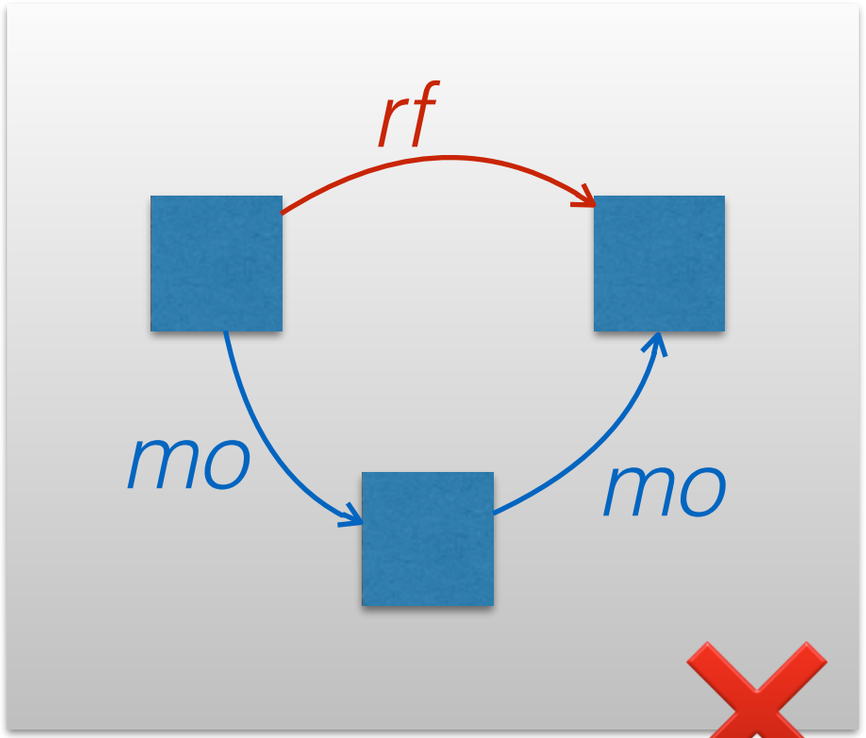


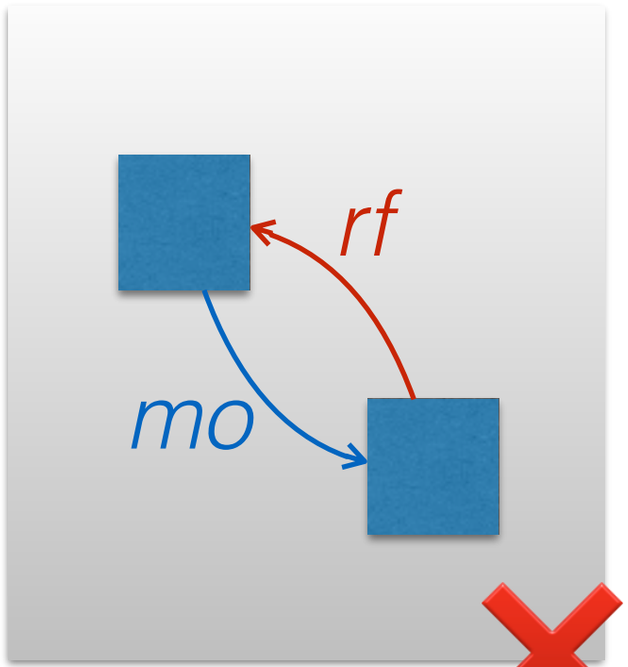


irreflexive(*rf*)

Some axioms

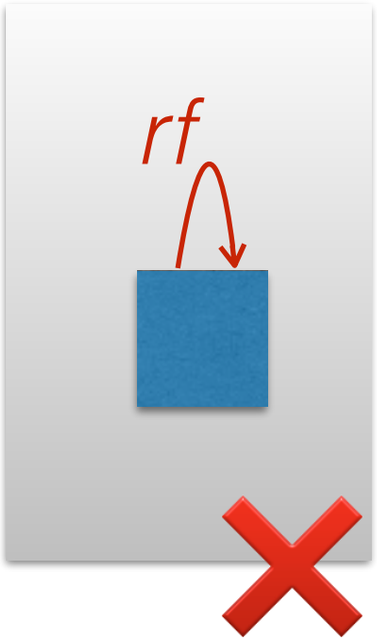




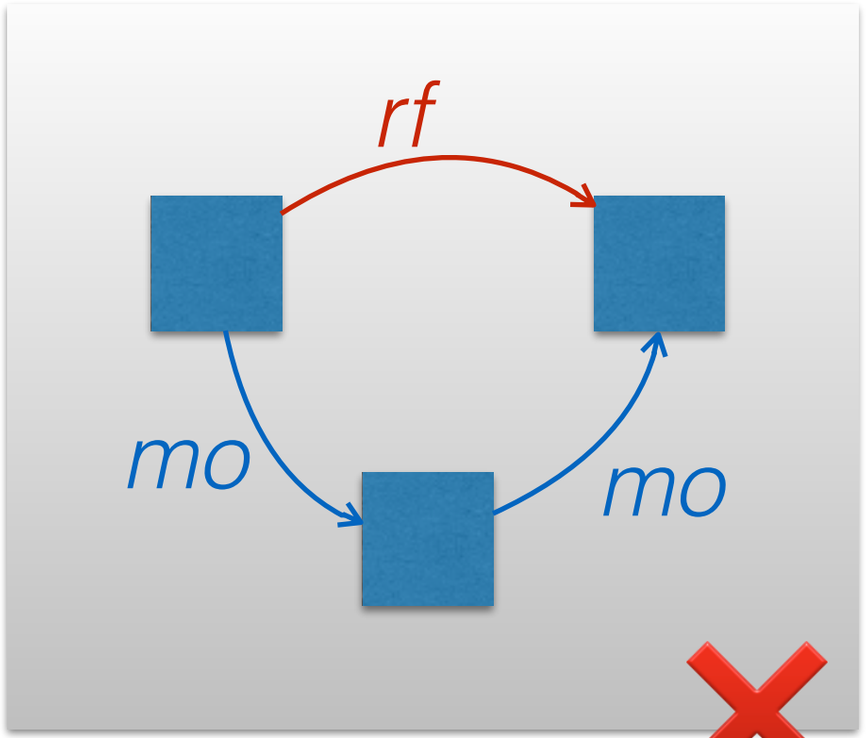


irreflexive(*rf*) irreflexive(*mo* ; *mo* ; rf^{-1})

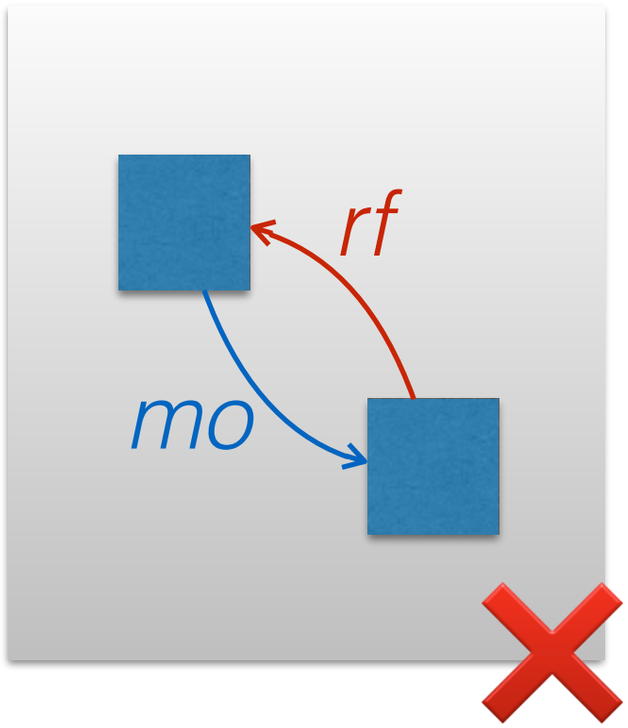
Some axioms



irreflexive(*rf*)



irreflexive(*mo* ; *mo* ; rf^{-1})



irreflexive(*mo* ; *rf*)

All consistency axioms

$$\mathrm{irr}(hb) \qquad \text{(Hb)}$$

$$\mathrm{irr}((rf^{-1})^{?} \,;\, mo \,;\, rf^{?} \,;\, hb) \qquad \text{(Coh)}$$

$$\mathrm{irr}(rf \,;\, hb) \qquad \text{(Rf)}$$

$$\mathrm{empty}((rf \,;\, [nal]) \setminus vis) \qquad \text{(NaRf)}$$

$$\mathrm{irr}(rf \cup (mo \,;\, mo \,;\, rf^{-1}) \cup (mo \,;\, rf)) \qquad \text{(Rmw)}$$

$$\mathrm{irr}(S \,;\, r_1) \quad \text{where } r_1 = hb \qquad \text{(S1)}$$

$$\mathrm{irr}(S \,;\, r_2) \quad \text{where } r_2 = Fsb^{?} \,;\, mo \,;\, sbF^{?} \qquad \text{(S2)}$$

$$\mathrm{irr}(S \,;\, r_3) \quad \text{where } r_3 = rf^{-1} \,;\, [\mathtt{SC}] \,;\, mo \qquad \text{(S3)}$$

$$\mathrm{irr}((S \setminus (mo \,;\, S)) \,;\, r_4) \quad \text{where } r_4 = rf^{-1} \,;\, hbl \,;\, [W] \qquad \text{(S4)}$$

$$\mathrm{irr}(S \,;\, r_5) \quad \text{where } r_5 = Fsb \,;\, rb \qquad \text{(S5)}$$

$$\mathrm{irr}(S \,;\, r_6) \quad \text{where } r_6 = rb \,;\, sbF \qquad \text{(S6)}$$

$$\mathrm{irr}(S \,;\, r_7) \quad \text{where } r_7 = Fsb \,;\, rb \,;\, sbF \qquad \text{(S7)}$$

Derived relations

$$
\begin{aligned}
acq &\stackrel{\text{def}}{=} (\mathtt{ACQ} \cup \mathtt{AR} \cup \mathtt{SC}) \cap (R \cup F) \\
rel &\stackrel{\text{def}}{=} (\mathtt{REL} \cup \mathtt{AR} \cup \mathtt{SC}) \cap (W \cup F) \\
rb &\stackrel{\text{def}}{=} (rf^{-1} \,;\, mo) \setminus id \\
Fsb &\stackrel{\text{def}}{=} [F] \,;\, sb \\
sbF &\stackrel{\text{def}}{=} sb \,;\, [F] \\
rs' &\stackrel{\text{def}}{=} thd \cup (E^2 \,;\, [R \cap W]) \\
rs &\stackrel{\text{def}}{=} mo \cap rs' \setminus ((mo \setminus rs') \,;\, mo) \\
sw &\stackrel{\text{def}}{=} ([rel] \,;\, Fsb^? \,;\, [W \cap A] \,;\, rs^? \,;\, rf \,; \\
&\qquad [R \cap A] \,;\, sbF^? \,;\, [acq]) \setminus thd \\
hb &\stackrel{\text{def}}{=} (sb \cup (I \times \neg I) \cup sw)^+ \\
hbl &\stackrel{\text{def}}{=} hb \cap =_{loc} \\
vis &\stackrel{\text{def}}{=} (W \times R) \cap hbl \setminus (hbl \,;\, [W] \,;\, hb)
\end{aligned}
$$

Outline

- Introduction to the C11 memory model
- Overhauling the rules for SC atomics in C11
- Introduction to the OpenCL memory model
- Overhauling the rules for SC atomics in OpenCL

SC axioms

$$\mathrm{irr}(S\,;\,r_1) \qquad \text{where } r_1 = hb \qquad \text{(S1)}$$

$$\mathrm{irr}(S\,;\,r_2) \qquad \text{where } r_2 = Fsb^{?}\,;\,mo\,;\,sbF^{?} \qquad \text{(S2)}$$

$$\mathrm{irr}(S\,;\,r_3) \qquad \text{where } r_3 = rf^{-1}\,;\,[\mathtt{SC}]\,;\,mo \qquad \text{(S3)}$$

$$\mathrm{irr}((S \setminus (mo\,;\,S))\,;\,r_4) \qquad \text{where } r_4 = rf^{-1}\,;\,hbl\,;\,[W] \qquad \text{(S4)}$$

$$\mathrm{irr}(S\,;\,r_5) \qquad \text{where } r_5 = Fsb\,;\,rb \qquad \text{(S5)}$$

$$\mathrm{irr}(S\,;\,r_6) \qquad \text{where } r_6 = rb\,;\,sbF \qquad \text{(S6)}$$

$$\mathrm{irr}(S\,;\,r_7) \qquad \text{where } r_7 = Fsb\,;\,rb\,;\,sbF \qquad \text{(S7)}$$

SC axioms

$$\mathrm{irr}(S\ ;\ r_1) \qquad \text{where } r_1 = hb \qquad \text{(S1)}$$

$$\mathrm{irr}(S\ ;\ r_2) \qquad \text{where } r_2 = Fsb^{?}\ ;\ mo\ ;\ sbF^{?} \qquad \text{(S2)}$$

$$\mathrm{irr}(S\ ;\ r_3) \qquad \text{where } r_3 = rf^{-1}\ ;\ [\mathtt{SC}]\ ;\ mo \qquad \text{(S3)}$$

$$\mathrm{irr}(S\ ;\ r_4) \qquad \text{where } r_4 = rf^{-1}\ ;\ hbl\ ;\ [W] \qquad \text{(S4)}$$

$$\mathrm{irr}(S\ ;\ r_5) \qquad \text{where } r_5 = Fsb\ ;\ rb \qquad \text{(S5)}$$

$$\mathrm{irr}(S\ ;\ r_6) \qquad \text{where } r_6 = rb\ ;\ sbF \qquad \text{(S6)}$$

$$\mathrm{irr}(S\ ;\ r_7) \qquad \text{where } r_7 = Fsb\ ;\ rb\ ;\ sbF \qquad \text{(S7)}$$

SC axioms

$$\mathrm{irr}(S\ ;\ r_1) \qquad \text{where } r_1 = hb \qquad \text{(S1)}$$

$$\mathrm{irr}(S\ ;\ r_2) \qquad \text{where } r_2 = Fsb^?\ ;\ mo\ ;\ sbF^? \qquad \text{(S2)}$$

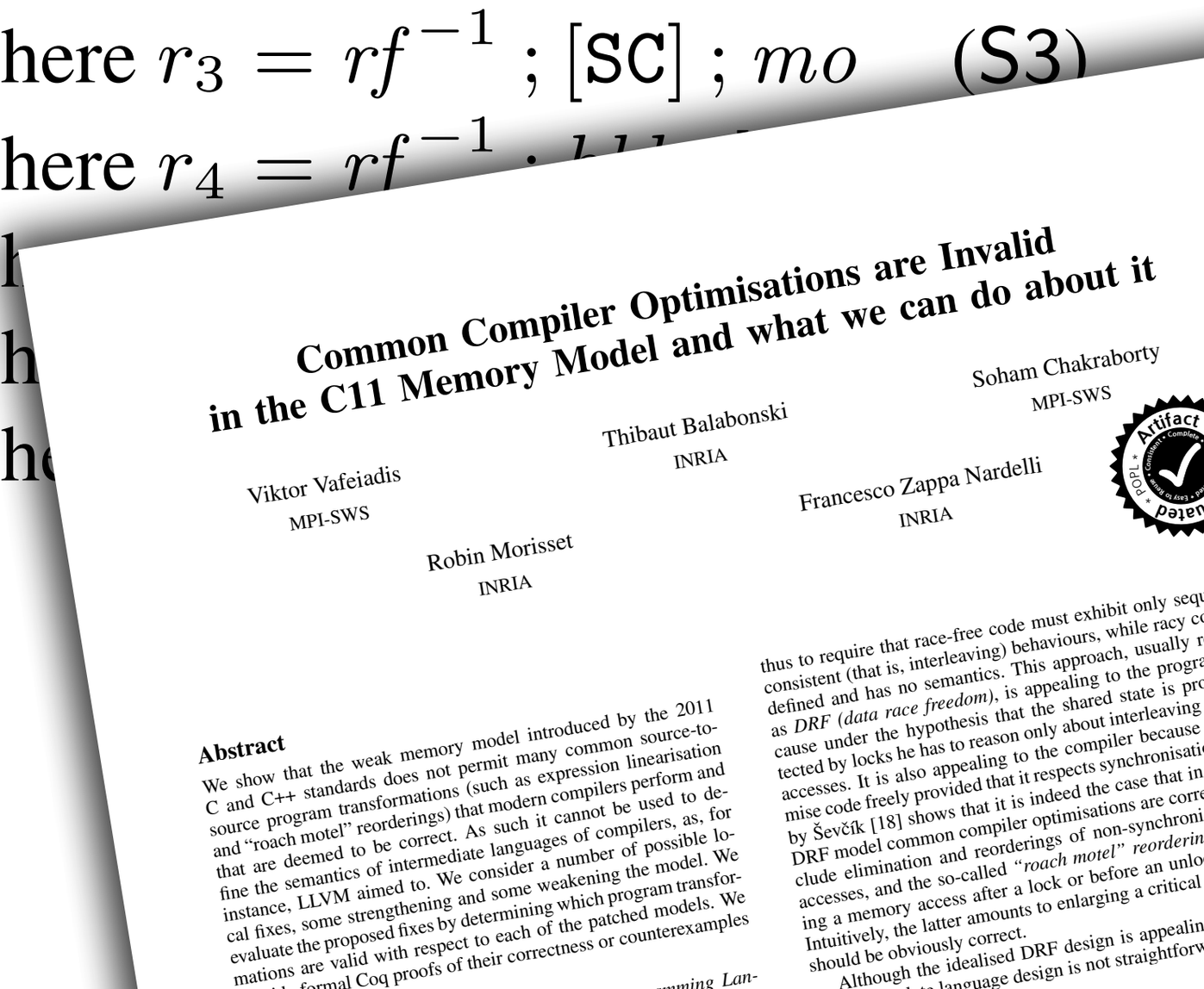
$$\mathrm{irr}(S\ ;\ r_3) \qquad \text{where } r_3 = rf^{-1}\ ;\ [\mathtt{SC}]\ ;\ mo \qquad \text{(S3)}$$

$$\mathrm{irr}(S\ ;\ r_4) \qquad \text{where } r_4 = rf^{-1}\ ;$$

$$\mathrm{irr}(S\ ;\ r_5)$$

$$\mathrm{irr}(S\ ;\ r_6)$$

$$\mathrm{irr}(S\ ;\ r_7)$$



Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it

Viktor Vafeiadis
MPI-SWS

Thibaut Balabonski
INRIA

Soham Chakraborty
MPI-SWS

Robin Morisset
INRIA

Francesco Zappa Nardelli
INRIA

Artifact Consistent Complete Well Documented Easy to Reuse POPL

Abstract

We show that the weak memory model introduced by the 2011 C and C++ standards does not permit many common source-to-source program transformations (such as expression linearisation and "roach motel" reorderings) that modern compilers perform and that are deemed to be correct. As such it cannot be used to define the semantics of intermediate languages of compilers, as, for instance, LLVM aimed to. We consider a number of possible local fixes, some strengthening and some weakening the model. We evaluate the proposed fixes by determining which program transformations are valid with respect to each of the patched models. We formal Coq proofs of their correctness or counterexamples

thus to require that race-free code must exhibit only sequ
consistent (that is, interleaving) behaviours, while racy co
defined and has no semantics. This approach, usually r
as *DRF* (*data race freedom*), is appealing to the progra
cause under the hypothesis that the shared state is pro
tected by locks he has to reason only about interleaving
accesses. It is also appealing to the compiler because
mise code freely provided that it respects synchronisati
by Ševčík [18] shows that it is indeed the case that in
DRF model common compiler optimisations are corre
clude elimination and reorderings of non-synchroni
accesses, and the so-called *"roach motel" reorderin*
ing a memory access after a lock or before an unlo
Intuitively, the latter amounts to enlarging a critical
should be obviously correct.
Although the idealised DRF design is appealin
language design is not straightforw

Consistent executions

- Execution *X* is **consistent** iff
  there exists *rf*, *mo* and *S* such that
    (*X*,*rf*,*mo*,*S*) is well-formed and
      satisfies all the **consistency axioms**.

- ⟦P⟧ = P's consistent executions*

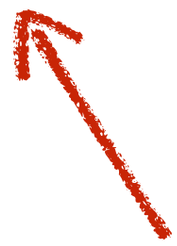
*unless P also admits a **faulty** execution, then ⟦P⟧ = any execution

SC axioms

$$\mathrm{acy}(\mathtt{SC}^2 \cap (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7) \setminus id) \quad (\mathsf{S}_{\mathsf{partial}})$$

SC axioms

$$\mathrm{acy}(\mathtt{SC}^2 \cap (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7) \setminus id) \quad (\mathsf{S}_{\mathsf{partial}})$$



This axiom is faster to simulate!

SC axioms

$$\mathrm{acy}(\mathtt{SC}^2 \cap (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7) \setminus id) \quad (\mathsf{S}_{\mathsf{partial}})$$

This axiom is faster to simulate!

Existing compilation schemes (x86 and Power) remain valid.

SC axioms

$$\mathrm{irr}(S\ ;\ r_1) \qquad \text{where } r_1 = hb \qquad \text{(S1)}$$

$$\mathrm{irr}(S\ ;\ r_2) \qquad \text{where } r_2 = Fsb^{?}\ ;\ mo\ ;\ sbF^{?} \qquad \text{(S2)}$$

$$\mathrm{irr}(S\ ;\ r_3) \qquad \text{where } r_3 = rf^{-1}\ ;\ [\mathtt{SC}]\ ;\ mo \qquad \text{(S3)}$$

$$\mathrm{irr}(S\ ;\ r_4) \qquad \text{where } r_4 = rf^{-1}\ ;\ hbl\ ;\ [W] \qquad \text{(S4)}$$

$$\mathrm{irr}(S\ ;\ r_5) \qquad \text{where } r_5 = Fsb\ ;\ rb \qquad \text{(S5)}$$

$$\mathrm{irr}(S\ ;\ r_6) \qquad \text{where } r_6 = rb\ ;\ sbF \qquad \text{(S6)}$$

$$\mathrm{irr}(S\ ;\ r_7) \qquad \text{where } r_7 = Fsb\ ;\ rb\ ;\ sbF \qquad \text{(S7)}$$

SC axioms

$$\mathrm{irr}(S\ ;\ r_1) \qquad \text{where } r_1 = hb \tag{S1}$$

$$\mathrm{irr}(S\ ;\ r_2) \qquad \text{where } r_2 = Fsb^{?}\ ;\ mo\ ;\ sbF^{?} \tag{S2}$$

$$\mathrm{irr}(S\ ;\ r_3) \qquad \text{where } r_3 = rf^{-1}\ ;\ mo \tag{S3}$$

$$\mathrm{irr}(S\ ;\ r_4) \qquad \text{where } r_4 = rf^{-1}\ ;\ hbl\ ;\ [W] \tag{S4}$$

$$\mathrm{irr}(S\ ;\ r_5) \qquad \text{where } r_5 = Fsb\ ;\ rb \tag{S5}$$

$$\mathrm{irr}(S\ ;\ r_6) \qquad \text{where } r_6 = rb\ ;\ sbF \tag{S6}$$

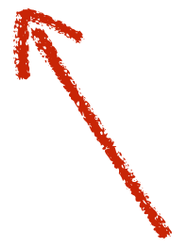
$$\mathrm{irr}(S\ ;\ r_7) \qquad \text{where } r_7 = Fsb\ ;\ rb\ ;\ sbF \tag{S7}$$

SC axioms

$$\mathrm{acy}(\mathtt{SC}^2 \cap (Fsb^? \,;\, (hb \cup rb \cup mo) \,;\, sbF^?)). \qquad (\mathsf{S_{simp}})$$

SC axioms

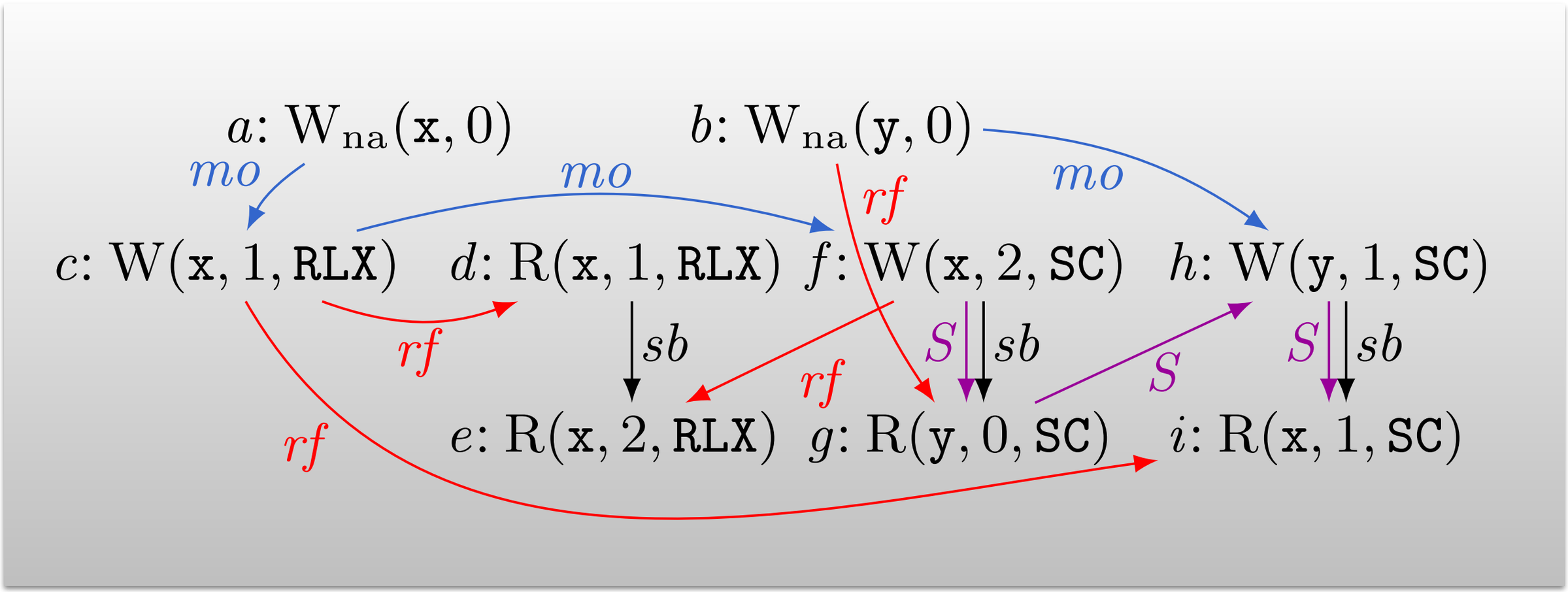
$$\mathrm{acy}(\mathtt{SC}^2 \cap (Fsb^? ; (hb \cup rb \cup mo) ; sbF^?)). \qquad (\mathsf{S_{simp}})$$



This axiom is much simpler for programmers to understand and to use

Existing compilation schemes (x86 and Power) remain valid.

Candidate executions



Changing the standard

6. There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:
   - the result of the last modification A of M that precedes B in S, if it exists, or
   - if A exists, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst` and that does not happen before A, or
   - if A does not exist, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst`.

[...]

9. For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.
10. For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.
11. For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.

[276 words; FK reading ease 41.2]

1. A value computation A of an object M *reads before* a side effect B on M if A and B are different operations and B follows, in the modification order of M, the side effect that A observes.
2. If X reads before Y, or happens before Y, or precedes Y in modification order, then X (as well as any fences sequenced before X) is *SC-before* Y (as well as any fences sequenced after Y).
3. If A is SC-before B, and A and B are both `memory_order_seq_cst`, then A is *restricted-SC-before* B.
4. There must be no cycles in restricted-SC-before.

[93 words; FK reading ease 73.1]

Outline

- Introduction to the C11 memory model
- Overhauling the rules for SC atomics in C11
- Introduction to the OpenCL memory model
- Overhauling the rules for SC atomics in OpenCL

OpenCL

- Execution hierarchy:

OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**

OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**

OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**

OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
    - Several devices form a **heterogeneous system**
- Memory hierarchy:

OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**
- Memory hierarchy:
  - `private` (accessible to one thread)

OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**
- Memory hierarchy:
  - `private` (accessible to one thread)
  - `local` (accessible to one work-group)

OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**
- Memory hierarchy:
  - `private` (accessible to one thread)
  - `local` (accessible to one work-group)
  - `global` (accessible to all devices)

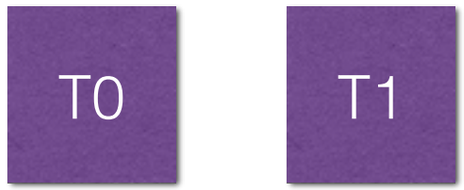
OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**
- Memory hierarchy:
  - `private` (accessible to one thread)
  - `local` (accessible to one work-group)
  - `global` (accessible to all devices)
  - `global_fga` (accessible to all devices, allows inter-device communication)

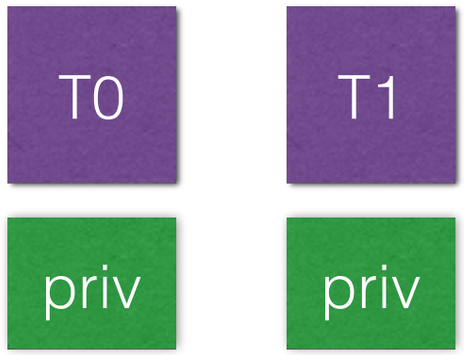
OpenCL memory regions

TO

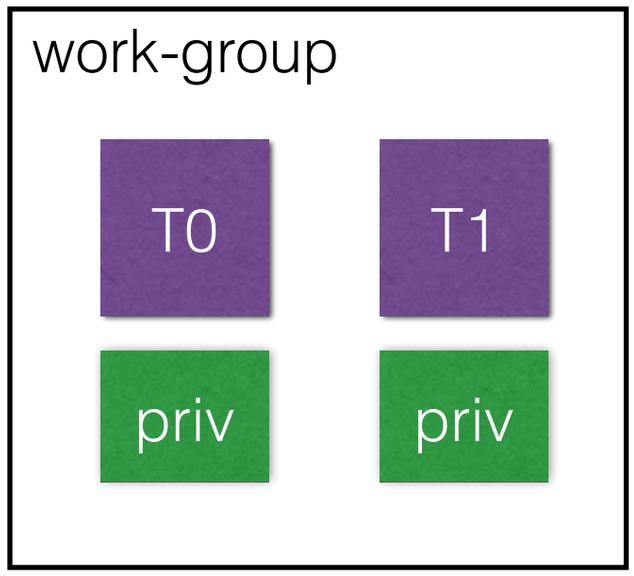
OpenCL memory regions



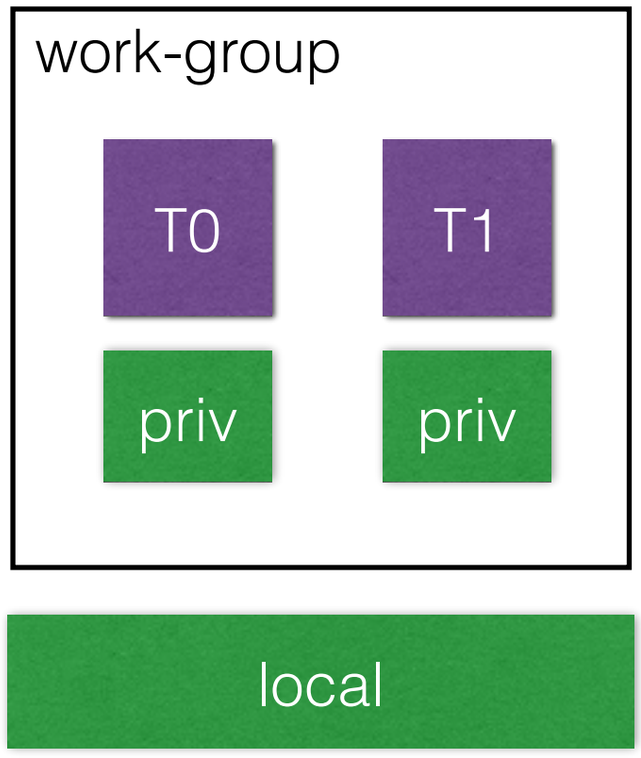
OpenCL memory regions



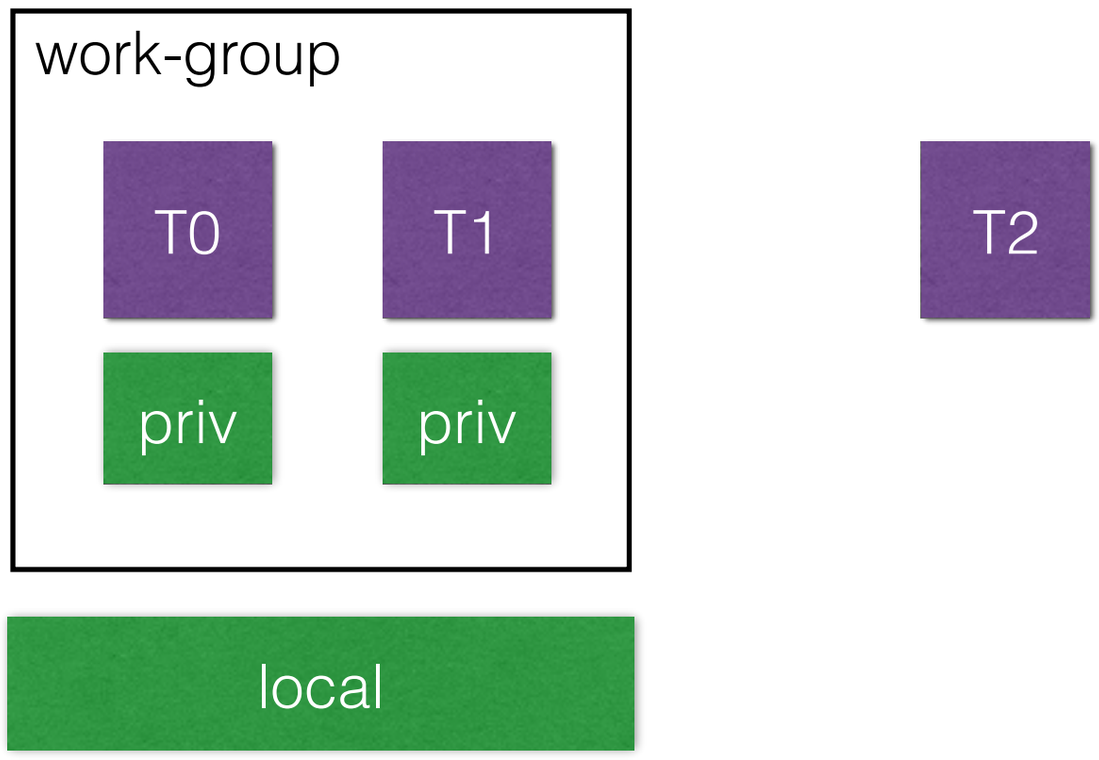
OpenCL memory regions



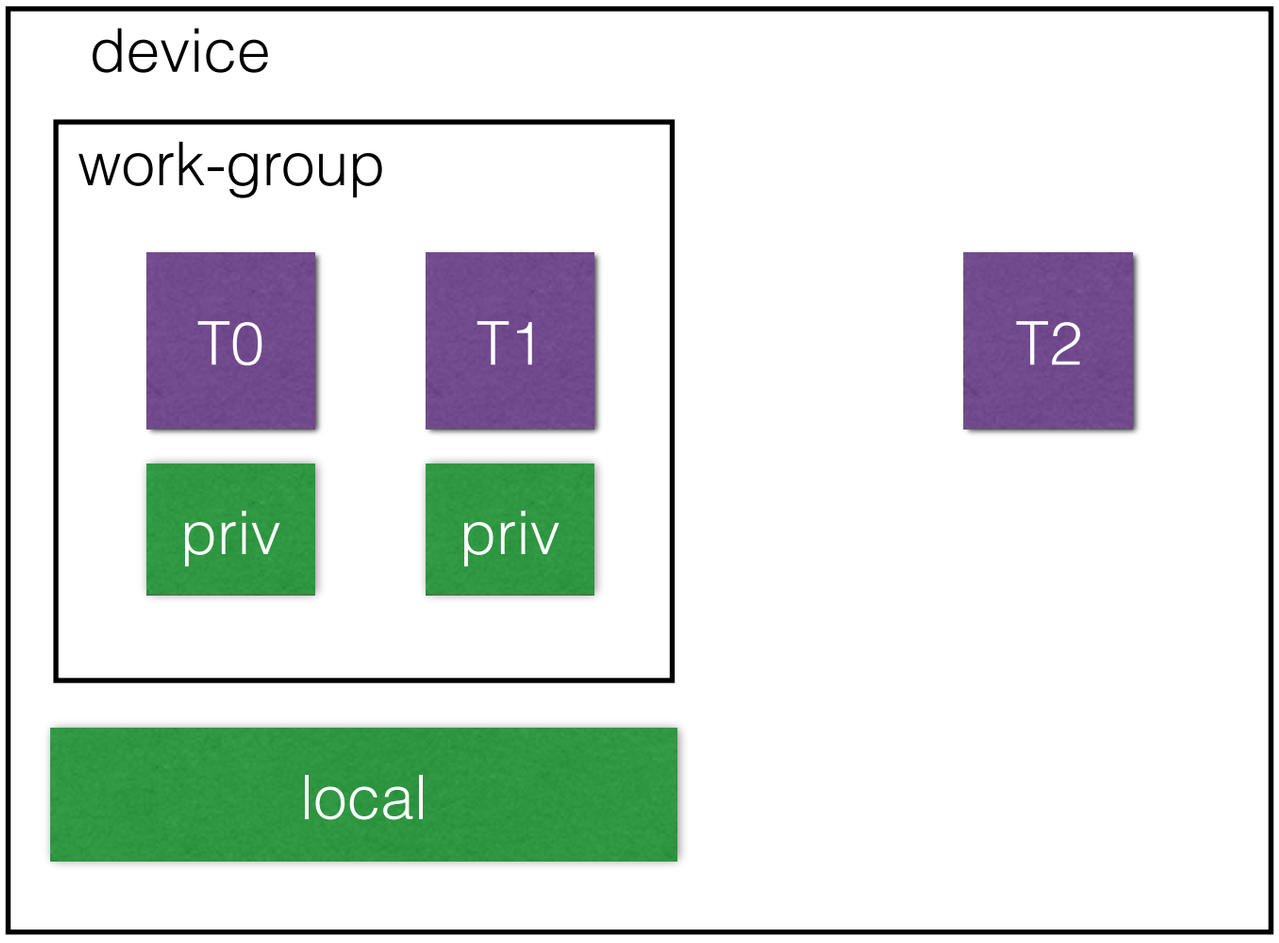
OpenCL memory regions



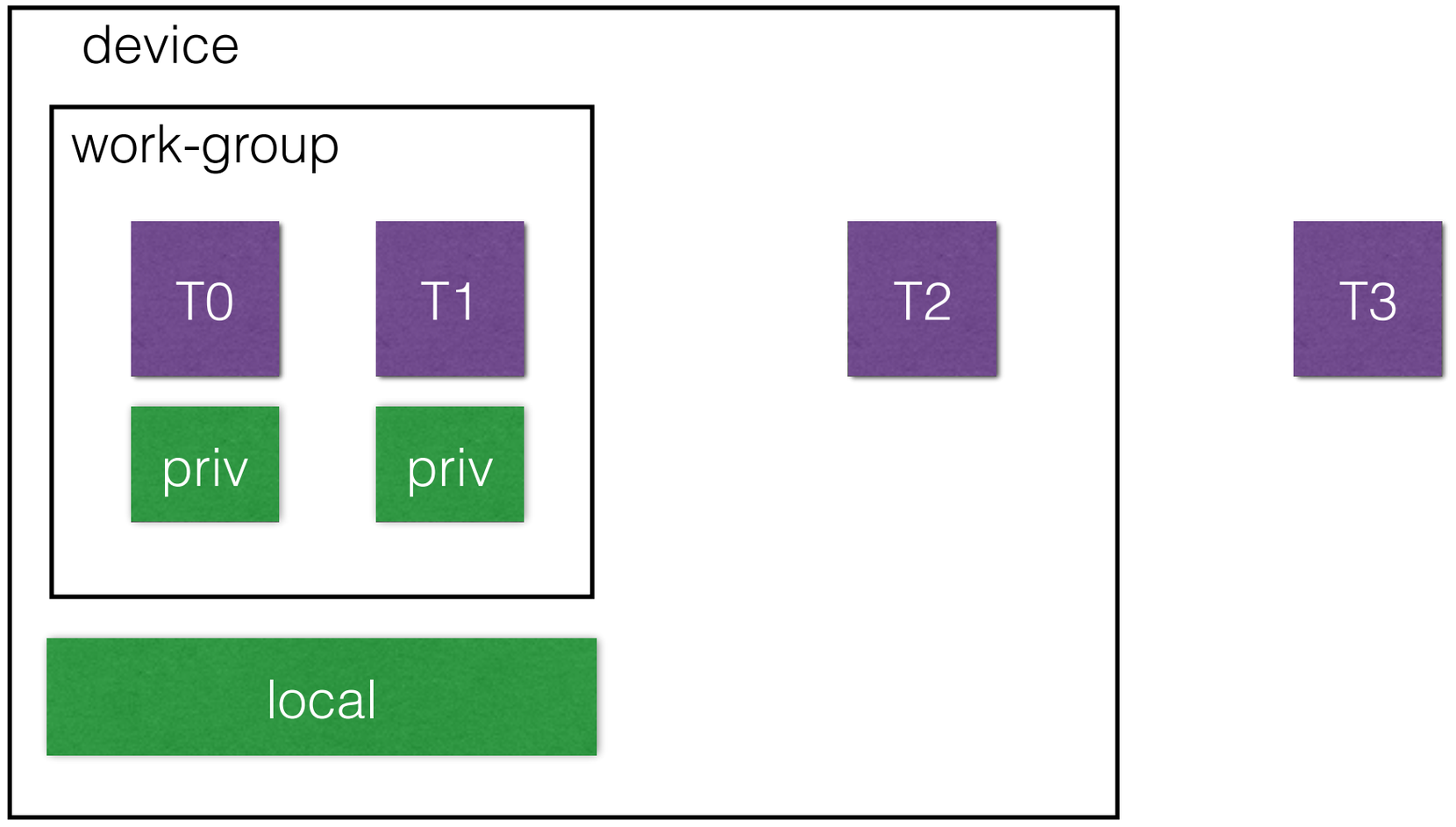
OpenCL memory regions



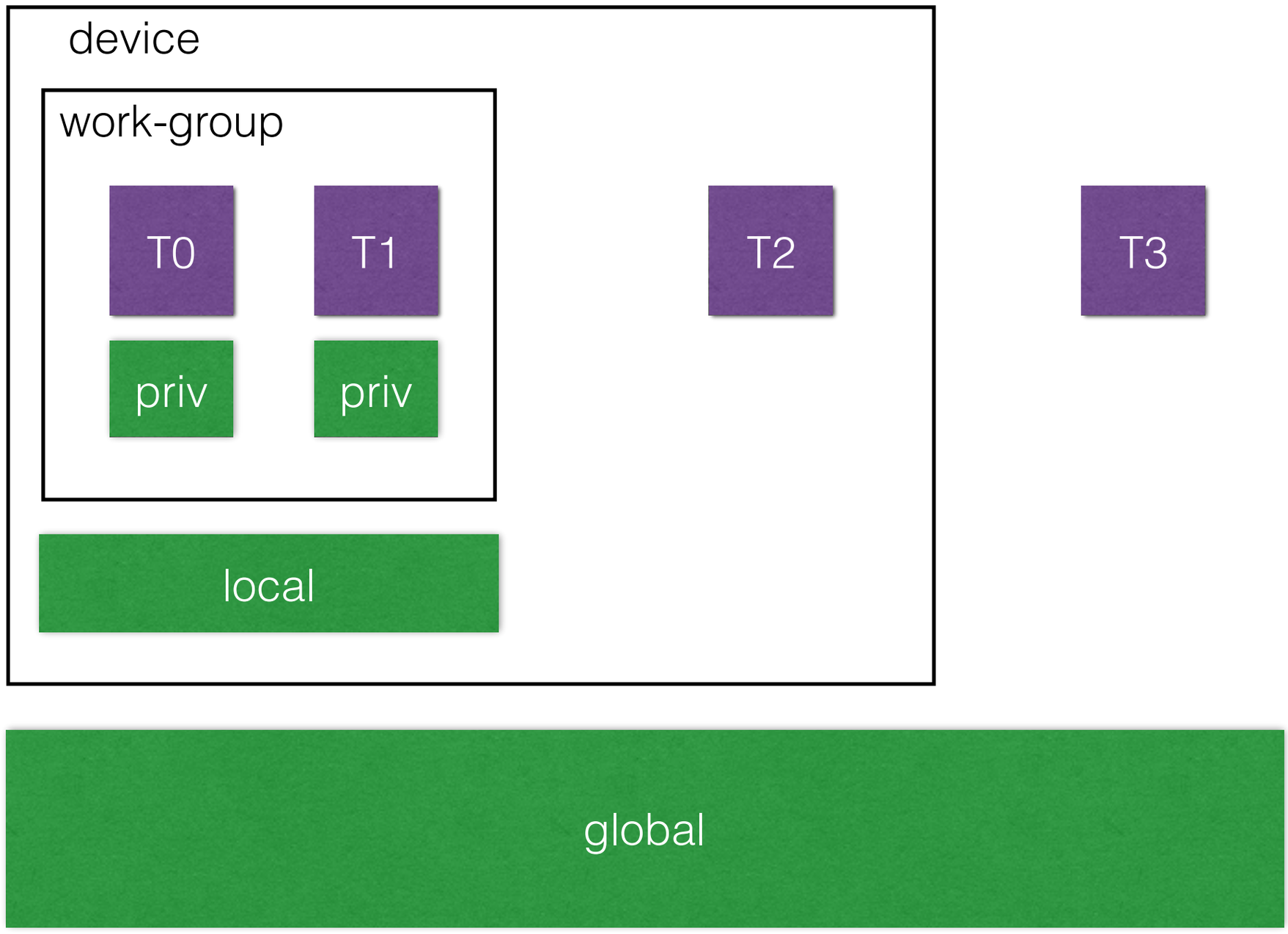
OpenCL memory regions



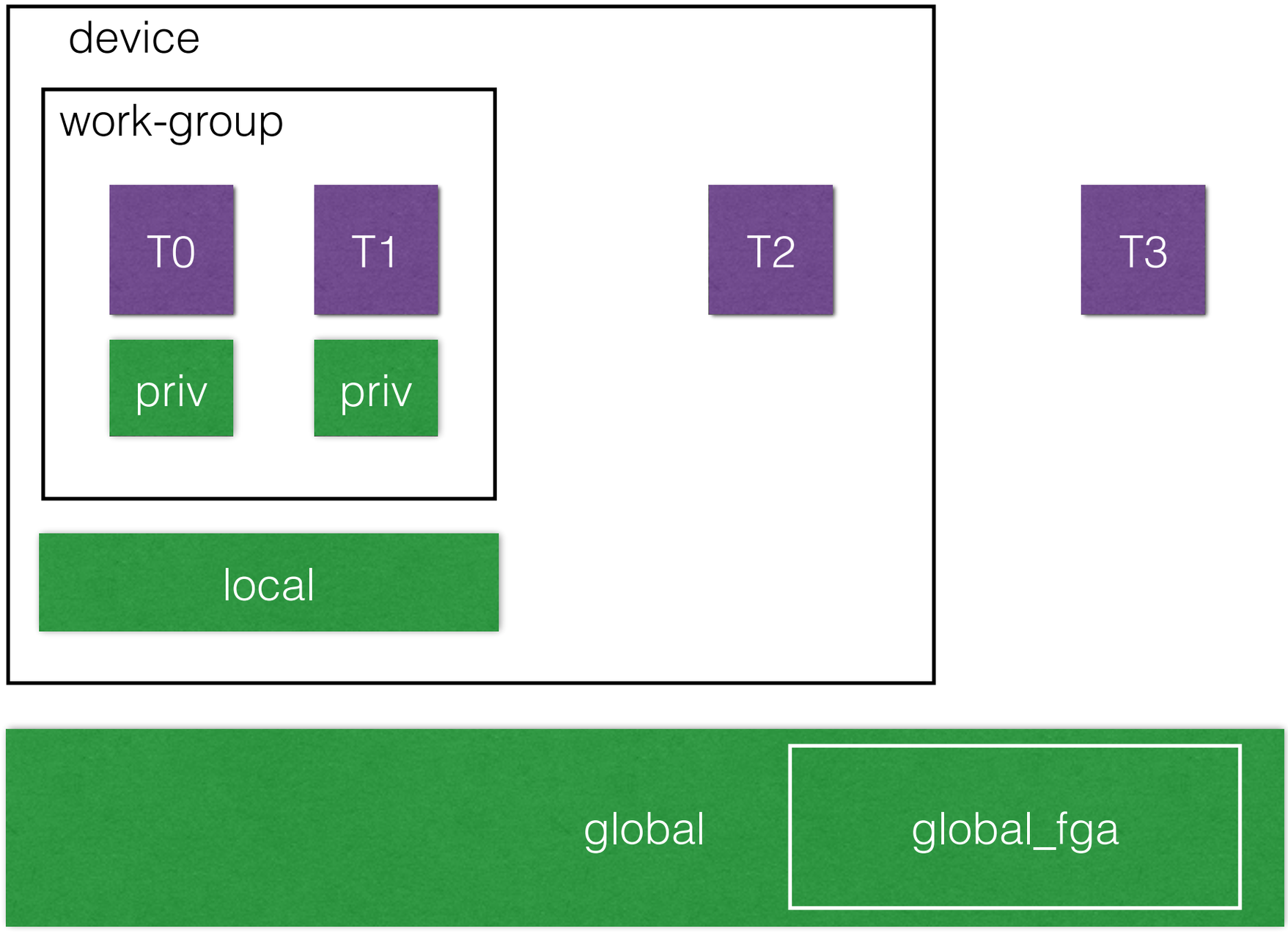
OpenCL memory regions



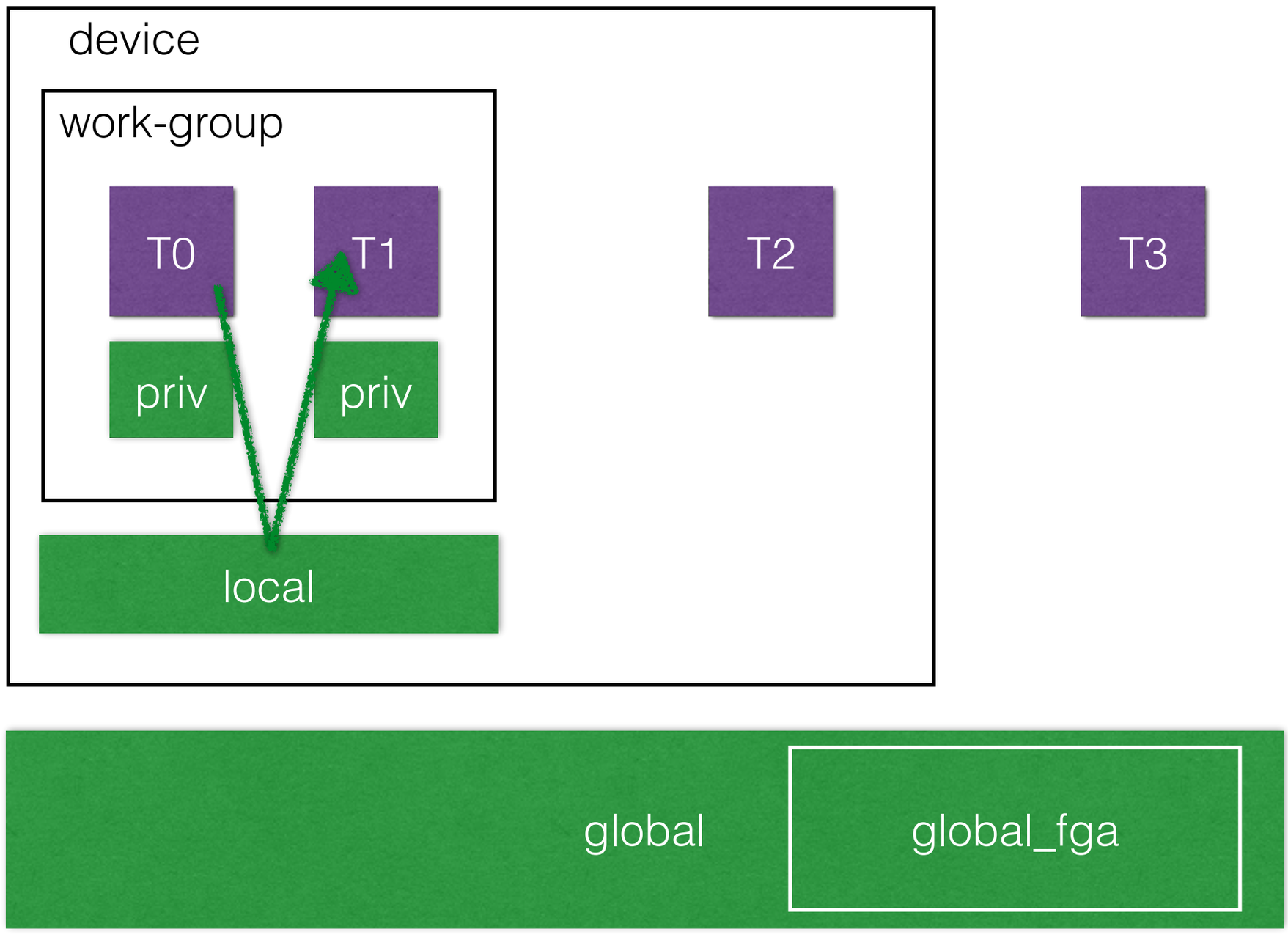
OpenCL memory regions



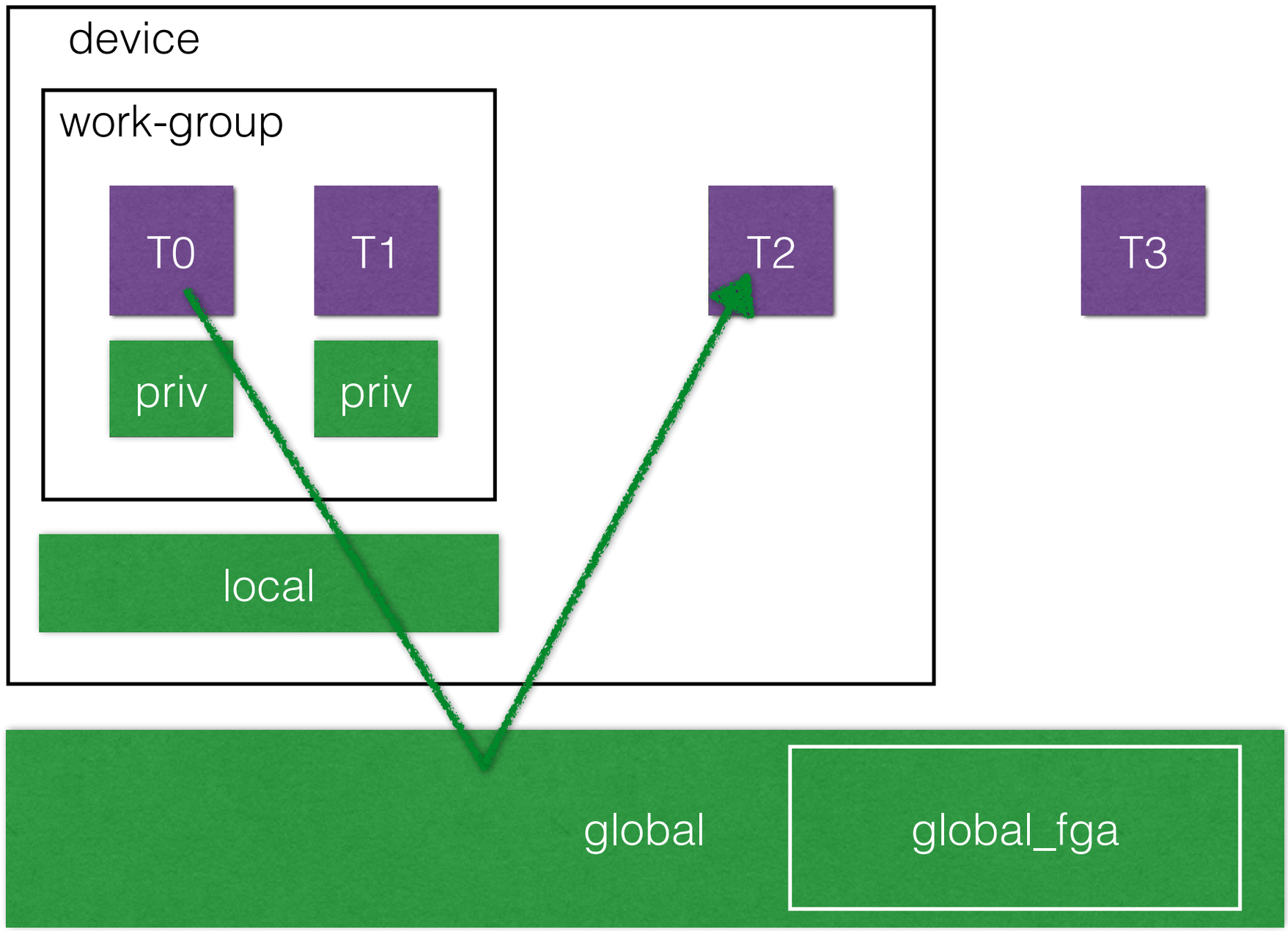
OpenCL memory regions



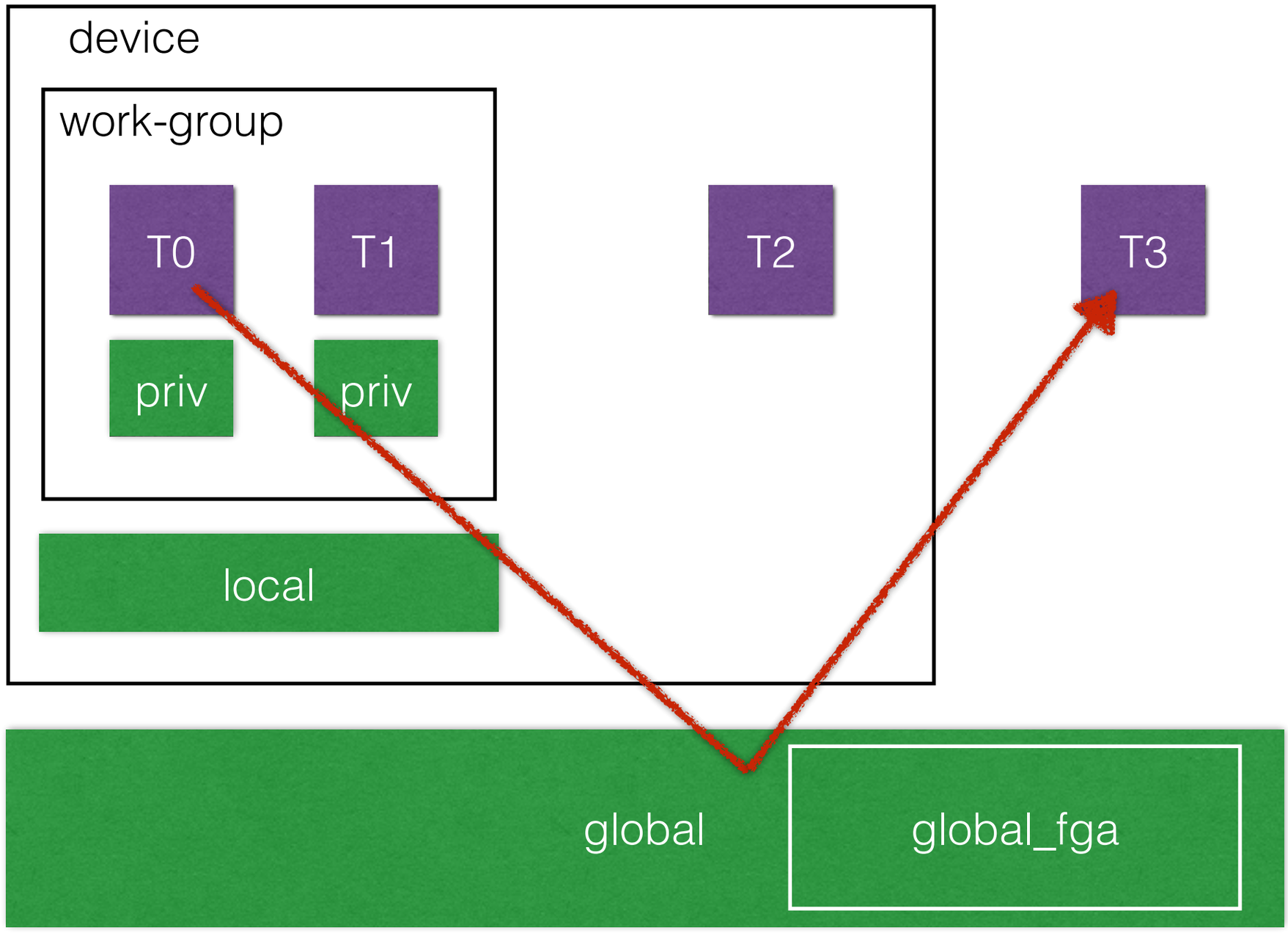
OpenCL memory regions



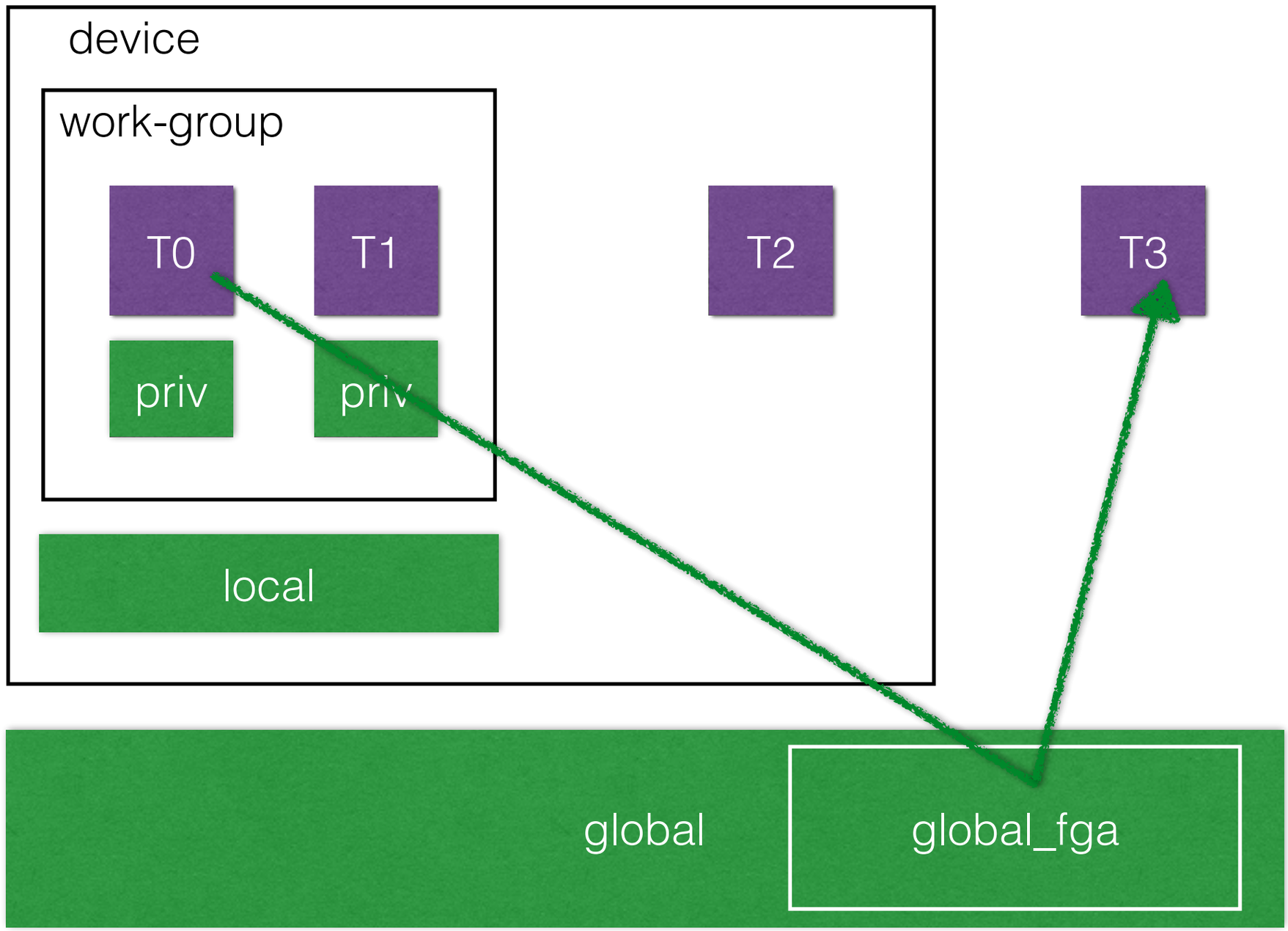
OpenCL memory regions



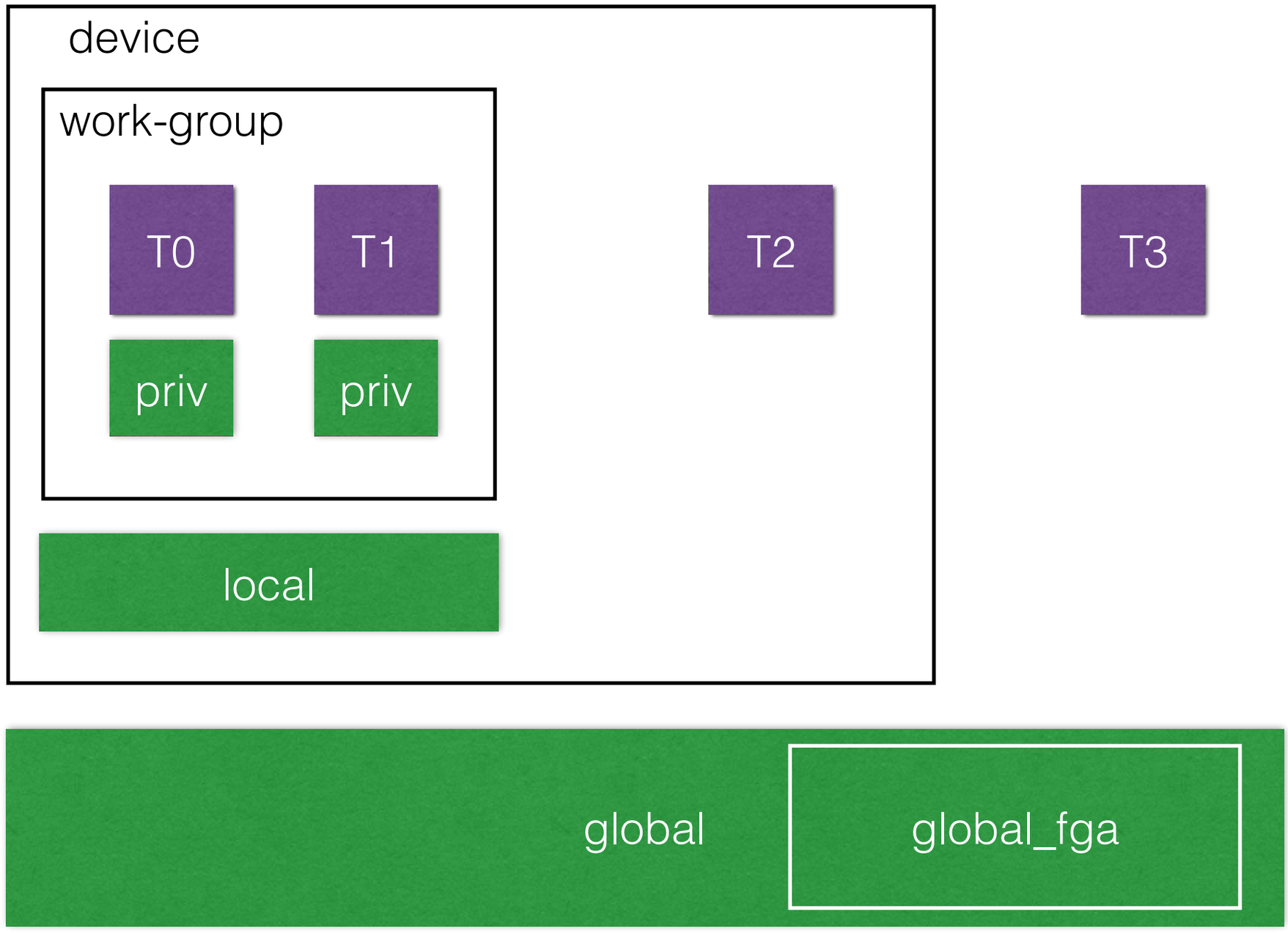
OpenCL memory regions



OpenCL memory regions



OpenCL memory regions



OpenCL memory scopes

- Memory consistency can be localised to one subtree of the execution hierarchy.

```
atomic_store_explicit(x, 1,
                      memory_order..., memory_scope_work_group );
```

OpenCL memory scopes

- Memory consistency can be localised to one subtree of the execution hierarchy.

```
atomic_store_explicit(x, 1,
                      memory_order..., memory_scope_device     );
                                       memory_scope_work_group
```

OpenCL memory scopes

- Memory consistency can be localised to one subtree of the execution hierarchy.

```
atomic_store_explicit(x, 1,
                      memory_order..., memory_scope_all_svm_devices);
                                       memory_scope_device
                                       memory_scope_work_group
```

Example

Threads in the same work-group

```
*x = 42;                                  ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,               ||      memory_order_acquire,
  memory_order_release,                   ||      memory_scope_work_group))
  memory_scope_work_group);               ||    print(x);
```

✓

Example

Threads in different work-groups, but same device

```
*x = 42;                                ||| if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,             |||     memory_order_acquire,
  memory_order_release,                 |||     memory_scope_work_group))
  memory_scope_work_group);             |||   print(x);
```

Example

Threads in different work-groups, but same device

```
*x = 42;                                ||| if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,             |||     memory_order_acquire,
  memory_order_release,                 |||     memory_scope_device))
  memory_scope_device);                 |||   print(x);
```

✓

Example

Threads in the same work-group

```
*x = 42;                              ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,           ||      memory_order_acquire,
  memory_order_release,               ||      memory_scope_device))
  memory_scope_device);               ||    print(x);
```

✓

Example

Threads in the same work-group

```
*x = 42;                                  ||  if (atomic_load_explicit(y,
atomic_store_explicit(y, 1,               ||      memory_order_acquire,
  memory_order_release,                   ||      memory_scope_work_group))
  memory_scope_device);                   ||    print(x);
```

Scope inclusion

- (*e1*,*e2*) ∈ *incl* iff:
  e1's scope is wide enough to reach *e2*, and
  e2's scope is wide enough to reach *e1*.

Outline

- Introduction to the C11 memory model
- Overhauling the rules for SC atomics in C11
- Introduction to the OpenCL memory model
- Overhauling the rules for SC atomics in OpenCL

SC axioms in OpenCL

SC axioms in OpenCL

- There is a total order *S* over [...]

SC axioms in OpenCL

- There is a total order *S* over [...]
- **PROVIDING** every SC operation has **`memory_scope_all_svm_devices`** and accesses **`global_fga`** memory

SC axioms in OpenCL

- There is a total order *S* over [...]
- **PROVIDING** every SC operation
  has **`memory_scope_all_svm_devices`** and
  accesses **`global_fga`** memory
- **OR** every SC operation
  has **`memory_scope_device`** and
  does not access **`global_fga`** memory

Problems

Problems

Can't always tell whether a location is `global` or `global_fga`!

Problems

- 😟 Can't always tell whether a location is `global` or `global_fga`!
- 😟 The default, which is `memory_scope_device`, is not always enough!

Problems

- 😟 Can't always tell whether a location is `global` or `global_fga`!
- 😟 The default, which is `memory_scope_device`, is not always enough!
- 😟 Non-compositional!

Problems

- 😟 Can't always tell whether a location is `global` or `global_fga`!
- 😟 The default, which is `memory_scope_device`, is not always enough!
- 😟 Non-compositional!
- 😟 Unnecessarily restrictive!

Problems

- 😟 Can't always tell whether a location is `global` or `global_fga`!
- 😟 The default, which is `memory_scope_device`, is not always enough!
- 😟 Non-compositional!
- 😟 Unnecessarily restrictive!
- 😟 And too weak anyway!

SC axioms in OpenCL

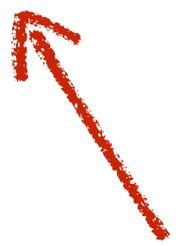
$$\mathrm{acy}(\mathtt{sc}^2 \cap (Fsb^? ; (ghb \cup lhb \cup rb \cup mo) ; sbF^?) \cap incl)$$

(O-S_{scoped})

SC axioms in OpenCL

$$\mathrm{acy}(\mathtt{sc}^2 \cap (Fsb^? ; (ghb \cup lhb \cup rb \cup mo) ; sbF^?) \cap incl)$$

(O-S_{scoped})



Existing compilation scheme
(for AMD GPUs) remains valid.

Changing the standard

If one of the following two conditions holds:

- All `memory_order_seq_cst` operations have the scope `memory_scope_all_svm_devices` and all affected memory locations are contained in system allocations or fine grain SVM buffers with atomics support
- All `memory_order_seq_cst` operations have the scope `memory_scope_device` and all affected memory locations are not located in system allocated regions or fine-grain SVM buffers with atomics support

then there shall exist a single total order S for all `memory_order_seq_cst` operations that is consistent with the modification orders for all affected locations, as well as the appropriate global-happens-before and local-happens-before orders for those locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M in global or local memory observes one of the following values:

- the result of the last modification A of M that precedes B in S, if it exists, or
- if A exists, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst` and that does not happen before A, or
- if A does not exist, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst`.

[...]
If the total order S exists, the following rules hold:

- For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced-before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.
- For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced-before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.
- For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced- before X, Y is sequenced-before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.
- For atomic operations A and B on an atomic object M, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced-before X, Y is sequenced-before B, and X precedes Y in S, then B occurs later than A in the modification order of M.

[391 words; FK reading ease -22.0]

1. A value computation A of an object M *reads before* a side effect B on M if A and B are different operations and B follows, in the modification order of M, the side effect that A observes.
2. If X reads before Y, or global happens before Y, or local happens before Y, or precedes Y in modification order, then X (as well as any fences sequenced before X) is *SC-before* Y (as well as any fences sequenced after Y).
3. If A is SC-before B, and A and B are both `memory_order_seq_cst`, and A and B have inclusive scopes, then A is *restricted-SC-before* B.
4. There must be no cycles in restricted-SC-before.

[106 words; FK reading ease 71.0]

TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are

  😕 too **complex**,

  😟 too **weak**, and

  😠 too **strong**.

- We suggest how to fix them 😌.