A dataflow model of concurrency, communication and weak memory

John Wickerson & Tony Hoare

Microsoft Research

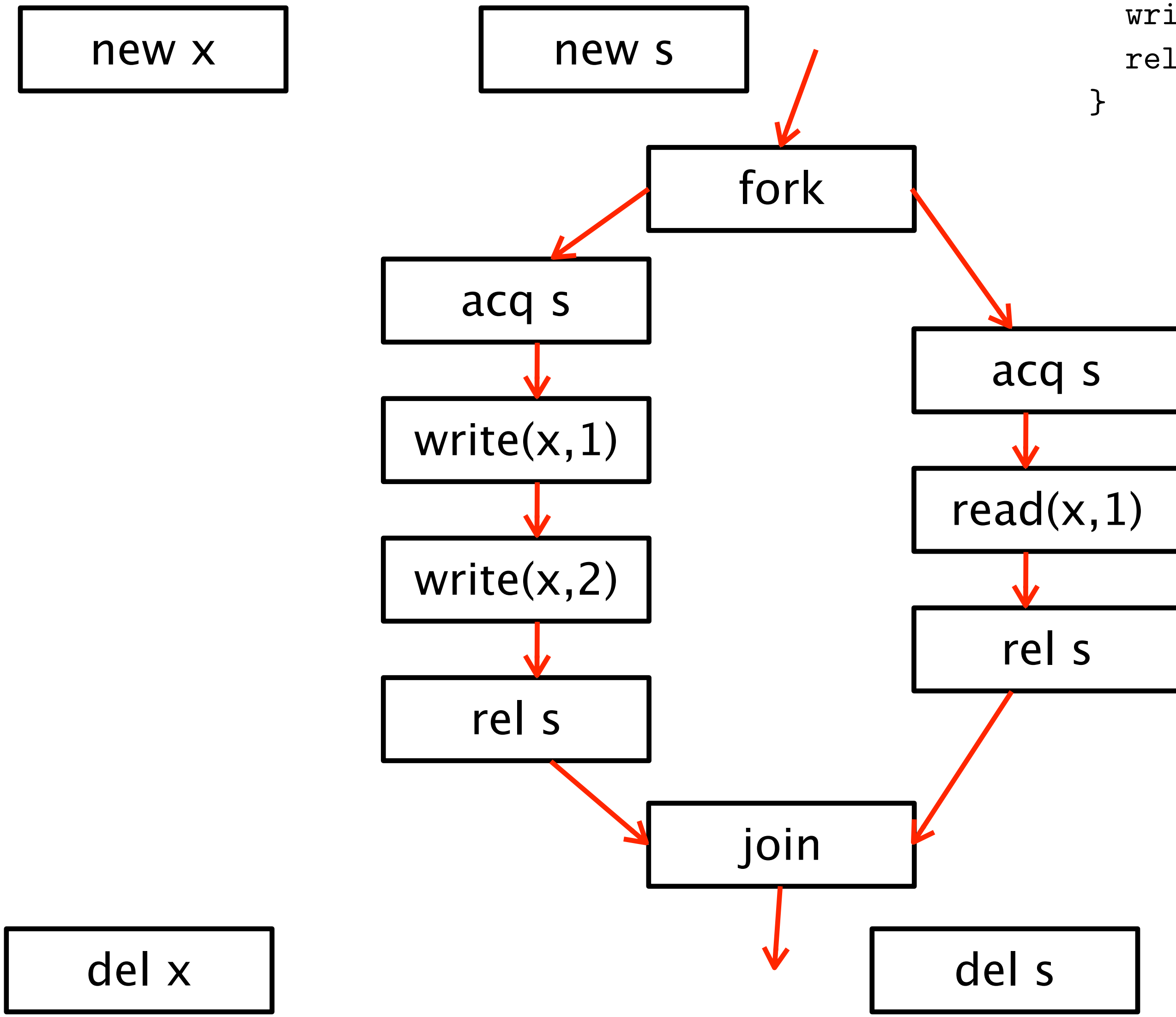
Cambridge Concurrency Workshop
5–6 July 2010

Example

```
lock s in var x in {
  acq s;          ∥ acq s;
  write(x,1);     ∥ read(x,1);
  write(x,2);     ∥ rel s
  rel s           ∥
}
```

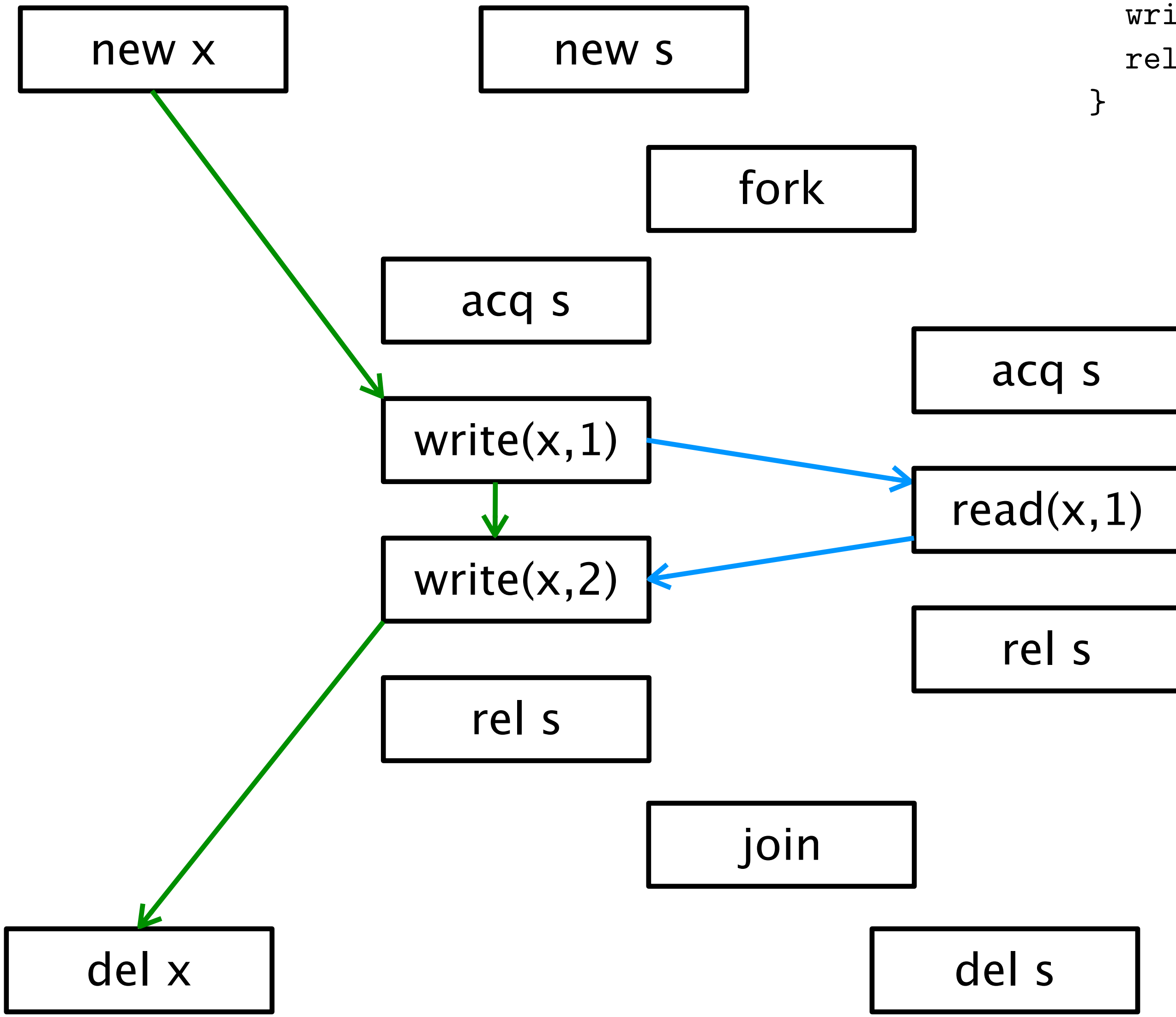
Example

```
lock s in var x in {
  acq s;       || acq s;
  write(x,1);  || read(x,1);
  write(x,2);  || rel s
  rel s
}
```



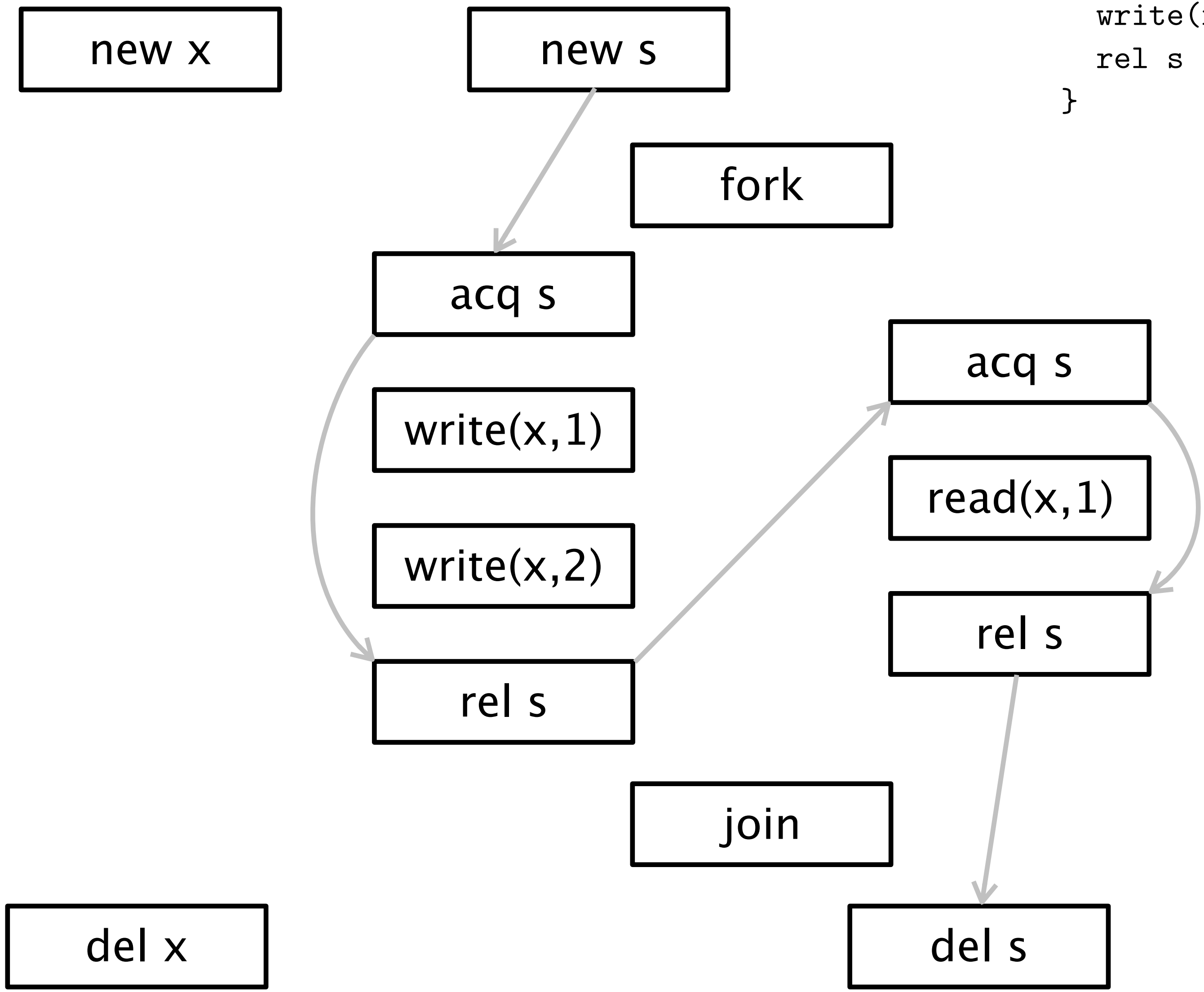
Example

```
lock s in var x in {
  acq s;         || acq s;
  write(x,1);    || read(x,1);
  write(x,2);    || rel s
  rel s          ||
}
```



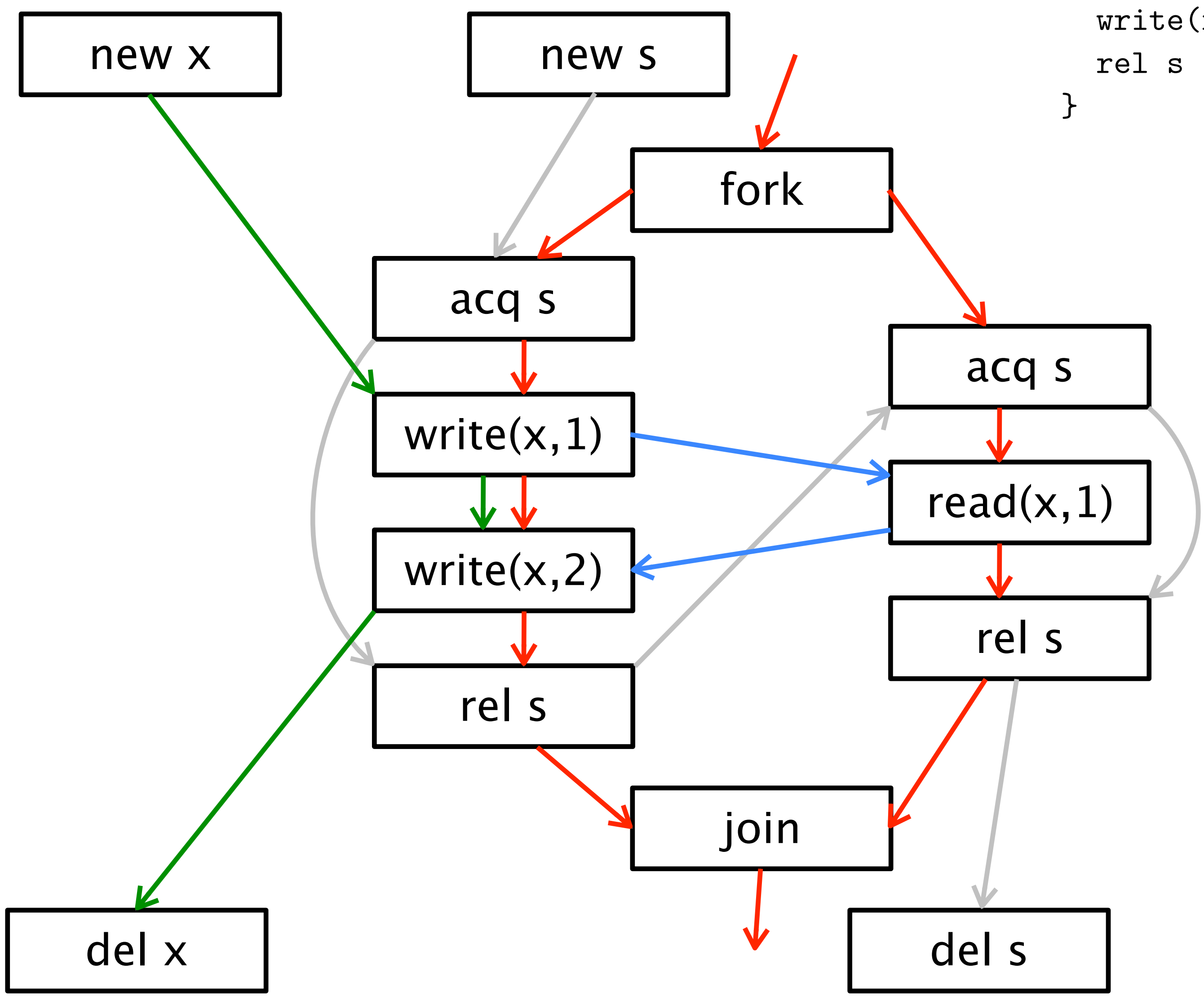
Example

```
lock s in var x in {
  acq s;        ||acq s;
  write(x,1);   || read(x,1);
  write(x,2);   || rel s
  rel s         ||
}
```



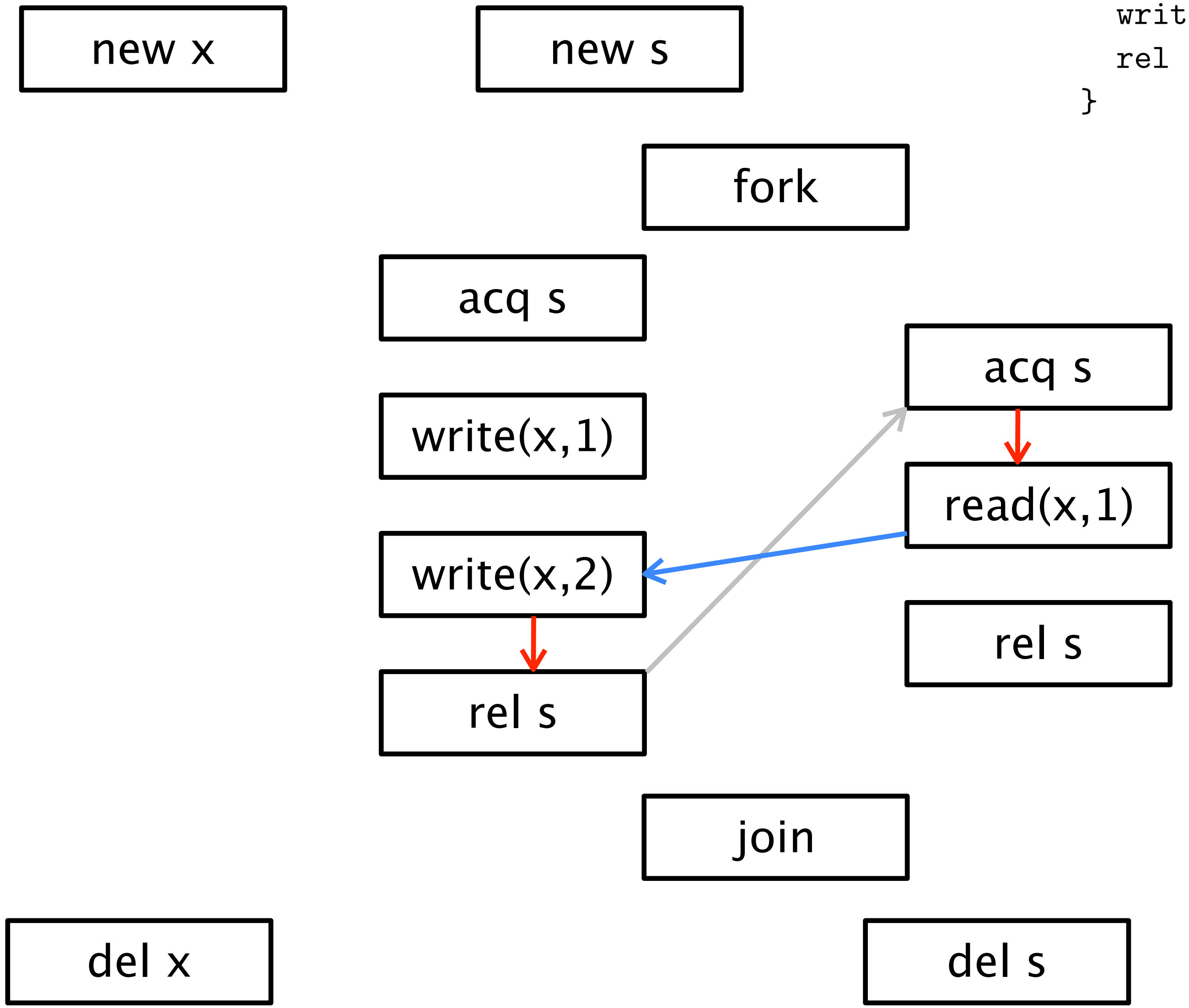
Example

```
lock s in var x in {
  acq s;          || acq s;
  write(x,1);     || read(x,1);
  write(x,2);     || rel s
  rel s
}
```



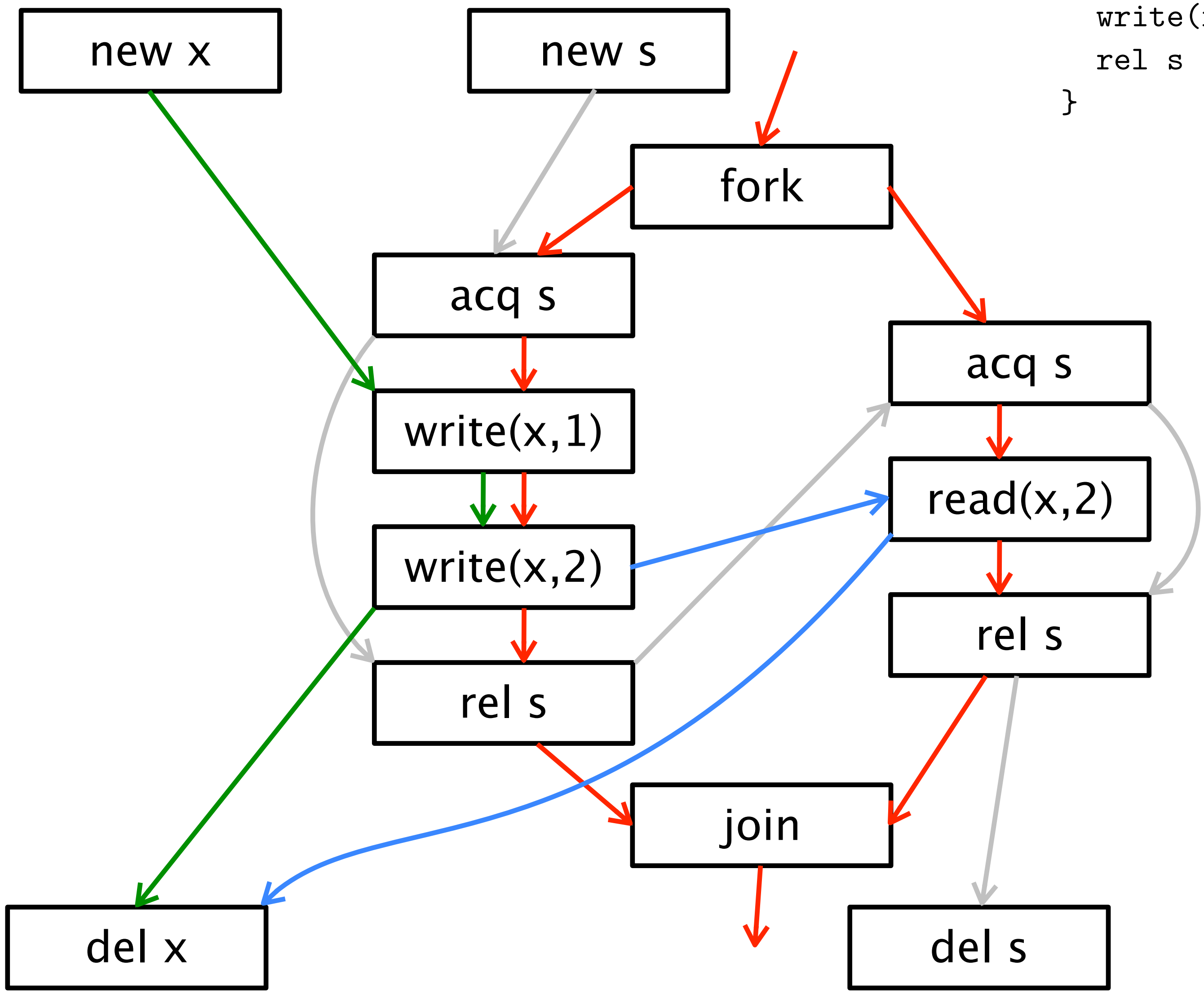
Example

```
lock s in var x in {
  acq s;        || acq s;
  write(x,1);   || read(x,1);
  write(x,2);   || rel s
  rel s
}
```



Example

```
lock s in var x in {
  acq s;        || acq s;
  write(x,1);   || read(x,2);
  write(x,2);   || rel s
  rel s
}
```



Outline

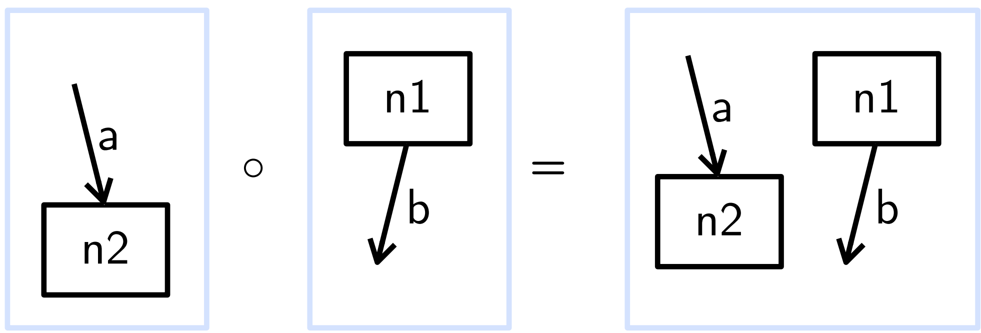
- We model a program as a set of possible traces
- We separate various kinds of flow
  - data flow, control flow, ownership transfer
- Our model is stateless
  - good for modelling weak memory and asynchronous communication

Traces

- Represented as a 6-tuple:
  - NodeSet, $N \in \mathbb{P}_{\text{fin}}\ \text{Node}$
  - ArrowSet, $A \in \mathbb{P}_{\text{fin}}\ \text{Arrow}$
  - Labelling, $L \in N \rightarrow \text{Label}$
  - Valuation, $V \in A \rightarrow \text{Value}$
  - HeadMap, $H \in A \rightharpoonup N$
  - TailMap, $T \in A \rightharpoonup N$

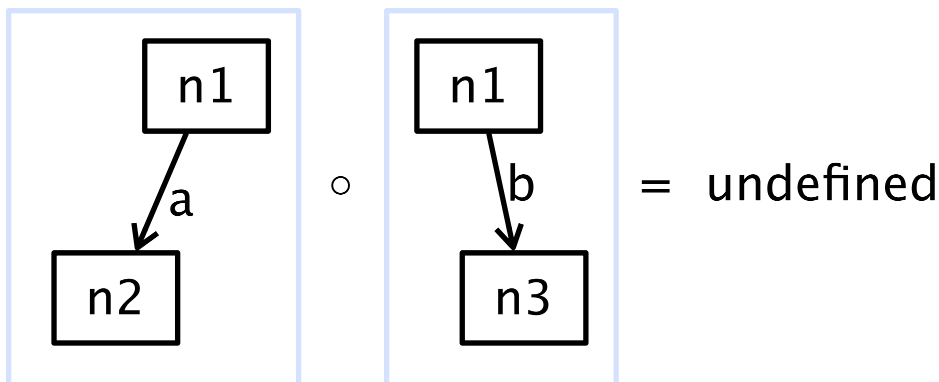
Traces

- A composition operator:



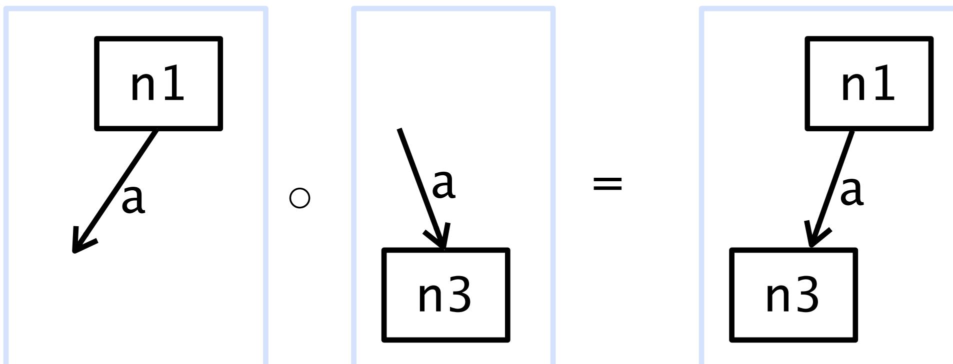
Traces

- A composition operator:



Traces

- A composition operator:



- Lifted to sets of traces:

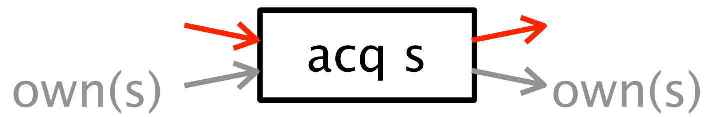
$$T * U = \{t \circ u \mid t \in T,\ u \in U\}$$

A denotational semantics

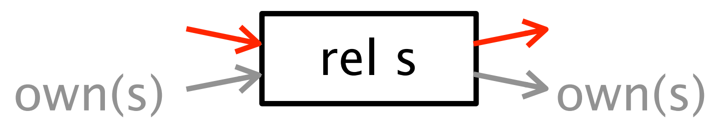
$$[\![-]\!] : \mathsf{Command} \to \mathbb{P}_{\mathsf{fin}}(\mathsf{Trace})$$

Locks

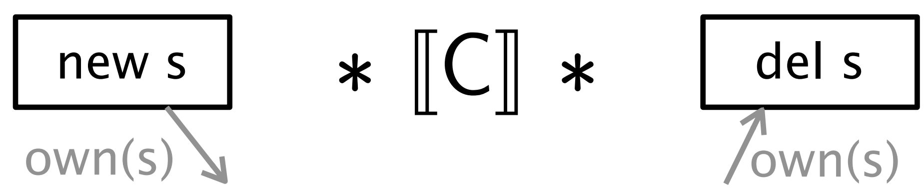
- C ::= … | lock s in C | acq s | rel s
- ⟦acq s⟧ =



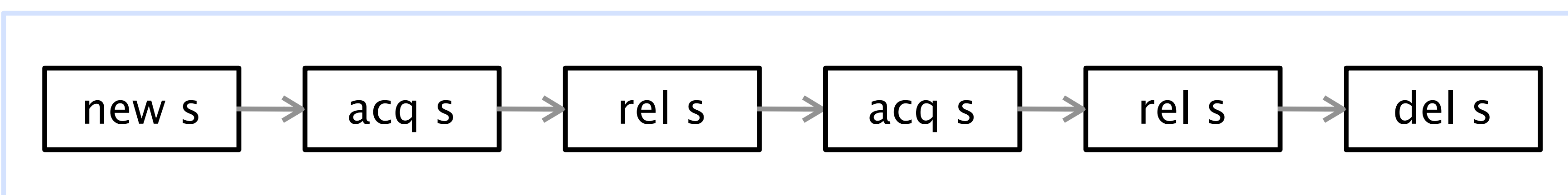
- ⟦rel s⟧ =



- ⟦lock s in C⟧ =

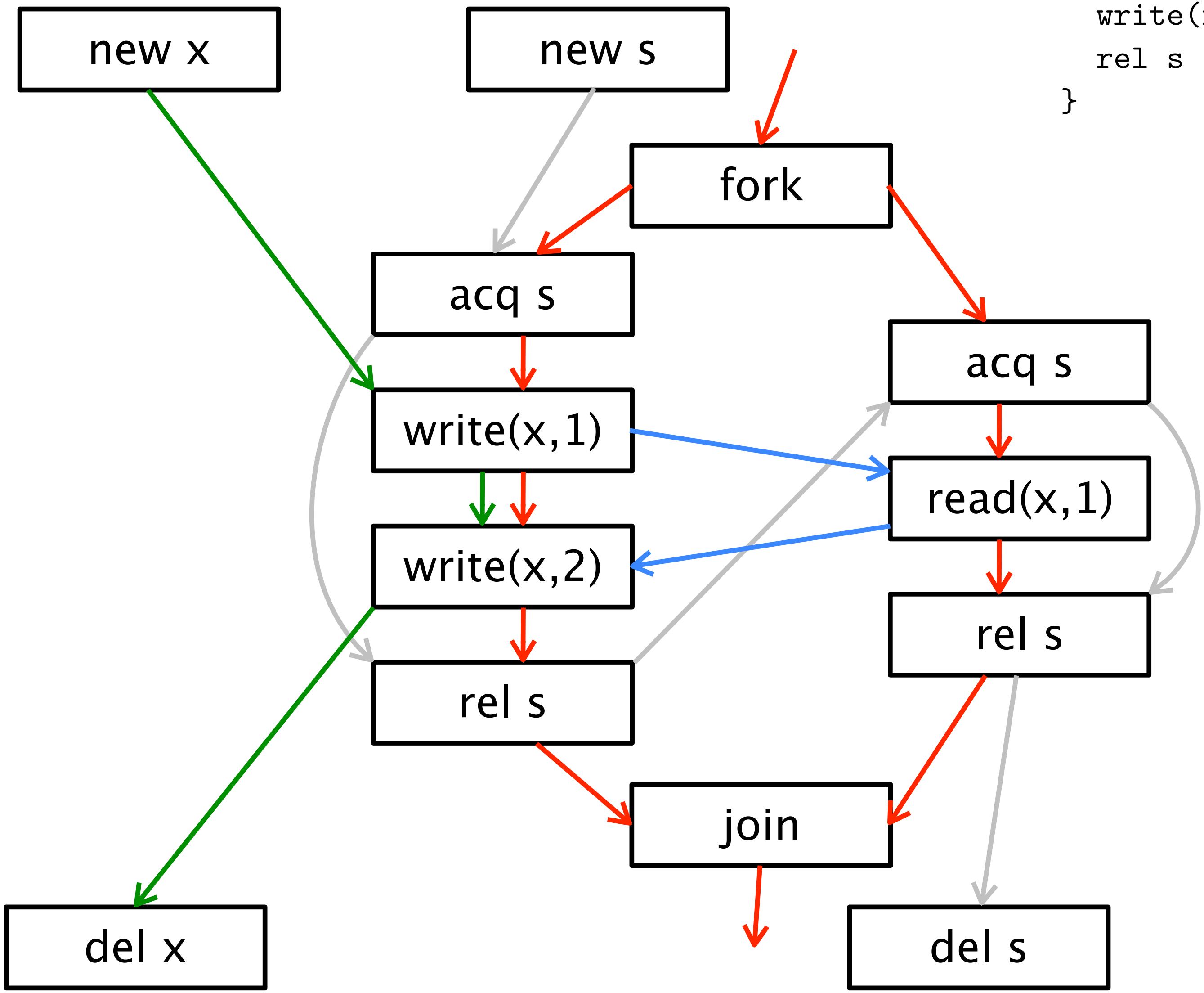


∩ lockconstraints(s)

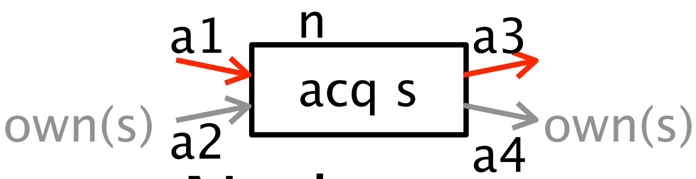


Example

```
lock s in var x in {
  acq s;        || acq s;
  write(x,1);   || read(x,1);
  write(x,2);   || rel s
  rel s         ||
}
```



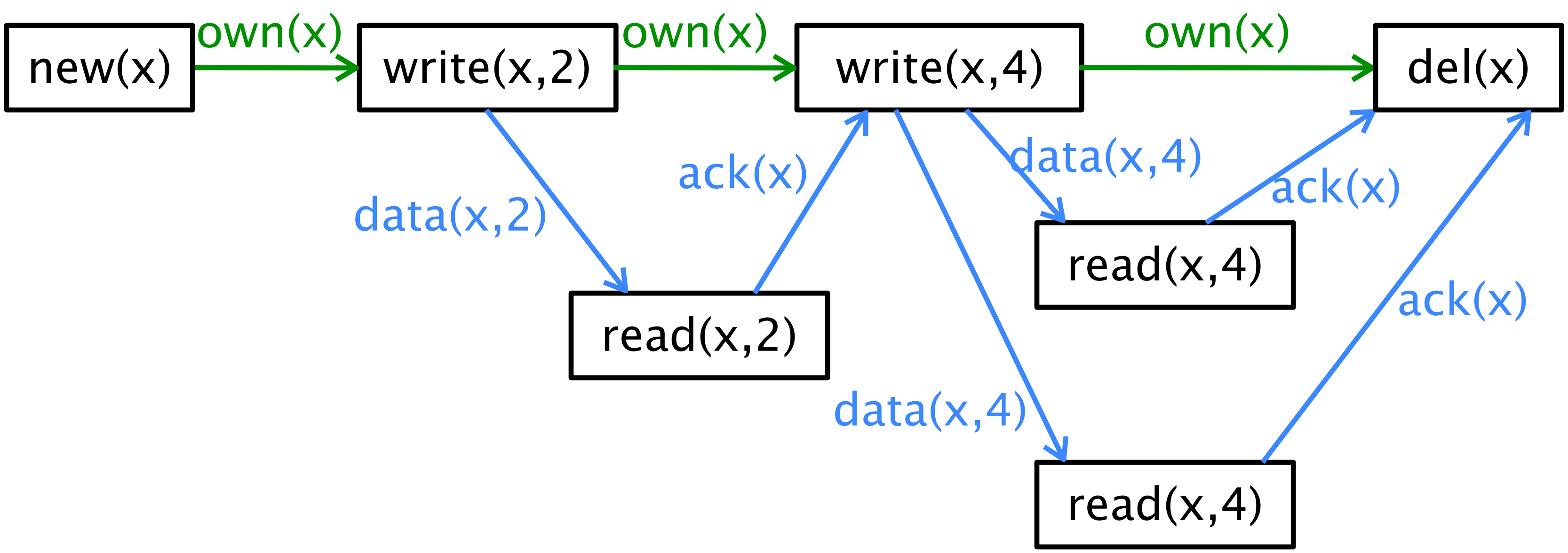
Locks

- $[\![\text{acq s}]\!] = \{$ 

 $|$
n ∈ Node,
a1,a2,a3,a4 ∈ Arrow,
a1,a2,a3,a4 all distinct }

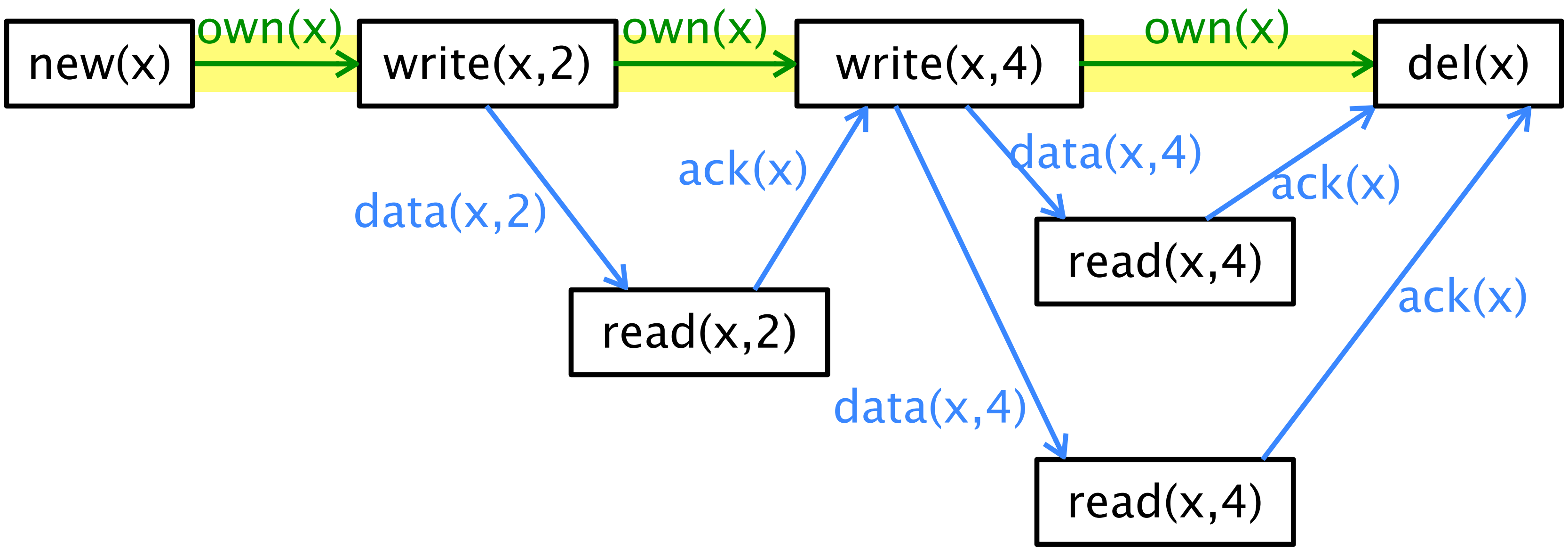
Variables

- C ::= … | var x in C | write(x,v) | read(x,v)



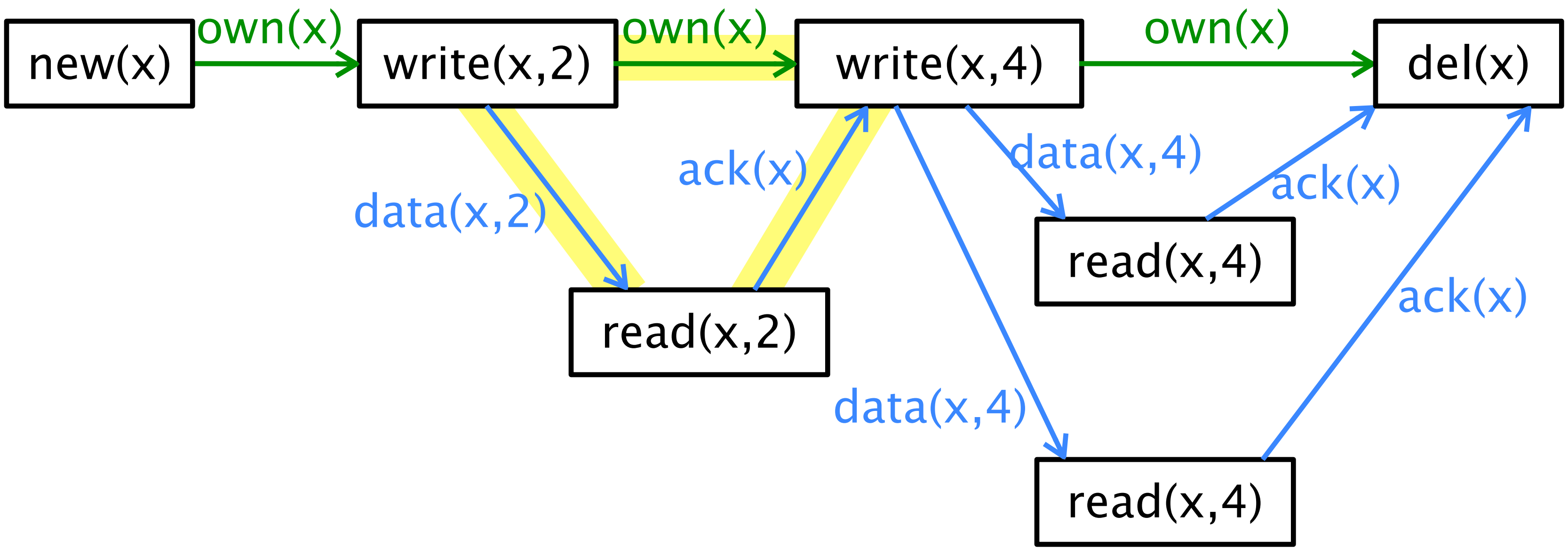
Variables

- C ::= … | var x in C | write(x,v) | read(x,v)



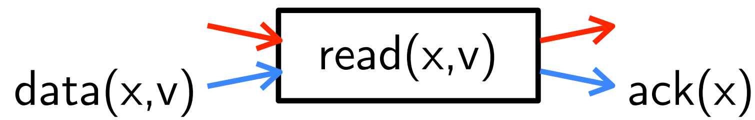
Variables

- C ::= … | var x in C | write(x,v) | read(x,v)



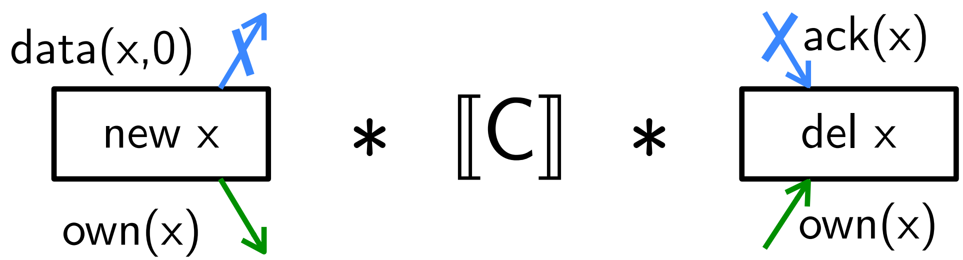
Variables

- C ::= … | var x in C | write(x,v) | read(x,v)
- $[\![\text{read}(x,v)]\!] =$



- $[\![\text{write}(x,v)]\!] =$

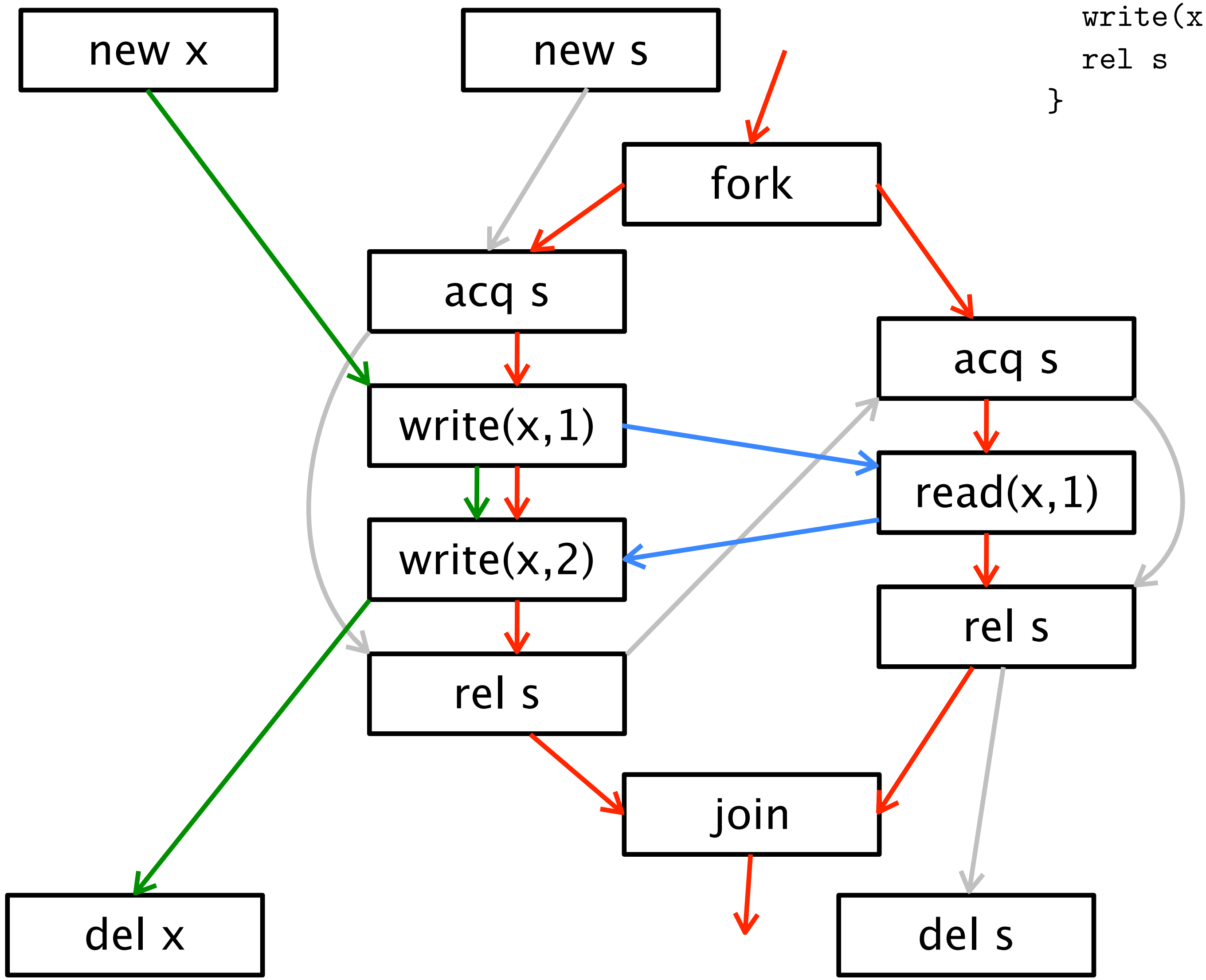
- $[\![\text{var x in C}]\!] =$



∩ varconstraints(x)

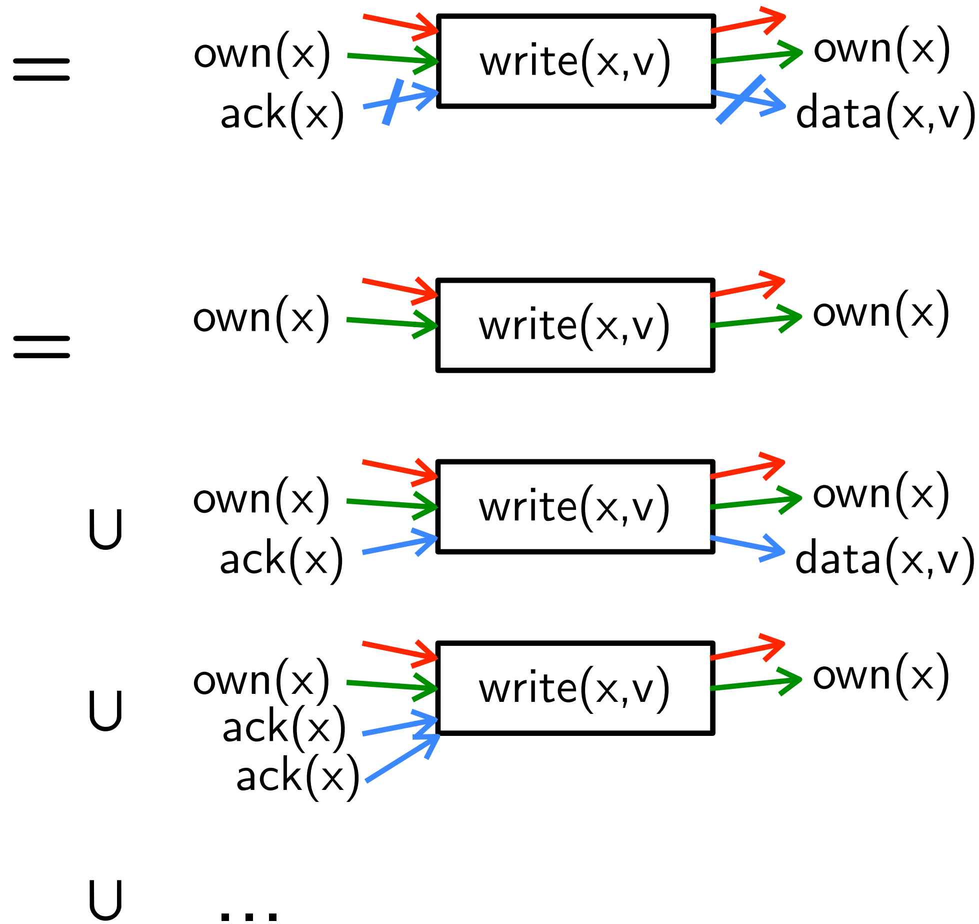
Example

```
lock s in var x in {
  acq s;        || acq s;
  write(x,1);   || read(x,1);
  write(x,2);   || rel s
  rel s         ||
}
```



Variables

- $[\![\text{write}(x,v)]\!]$



Assignments and assumptions

- $[\![x := f(y_1, \ldots, y_n)]\!] =$
  $\cup\{[\![read(y_1,v_1); \ldots ; read(y_n,v_n); write(x,v)]\!]$
  $\mid f(v_1,\ldots,v_n) = v \}$

- $[\![assume\ p(x_1, \ldots, x_n)]\!] =$
  $\cup\{[\![read(x_1,v_1); \ldots ; read(x_n,v_n)]\!]$
  $\mid p(v_1,\ldots,v_n) = true \}$

Sequential composition

- $[\![C_1;C_2]\!] = [\![C_1]\!] *_{seq} [\![C_2]\!]$

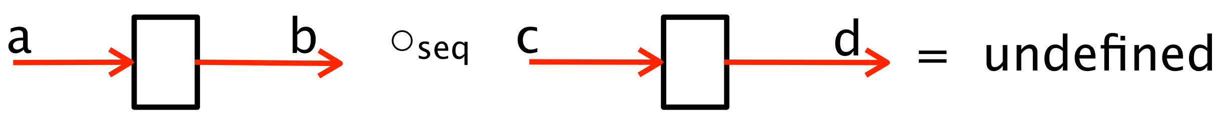
where $t_1 \circ_{seq} t_2$ is only defined when:

$$\mathrm{outCtrl}(t_1) = \mathrm{inCtrl}(t_2)$$

and $*_{seq}$ is the lifted version of $\circ_{seq}$

Sequential composition

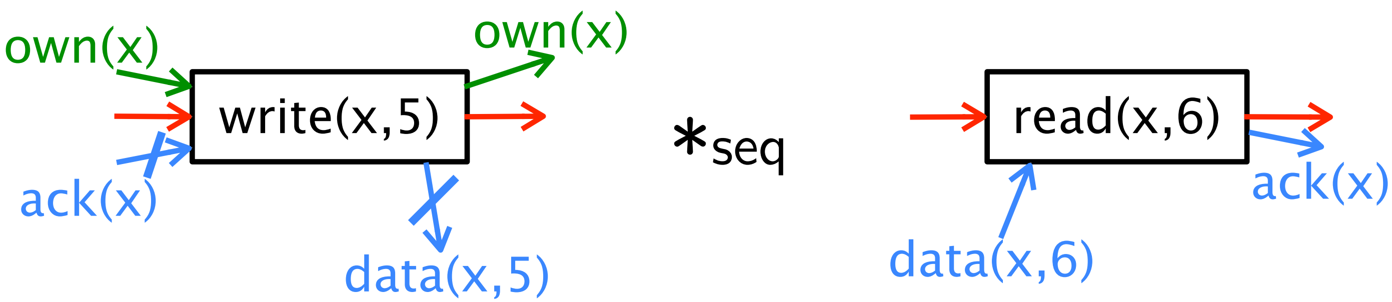
- Examples:



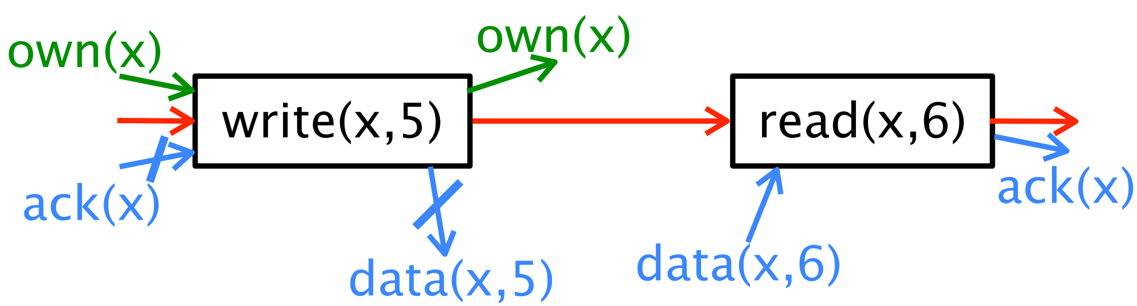
Sequential composition

- $[\![x:=5;\ \text{assume } x=6]\!]$

=

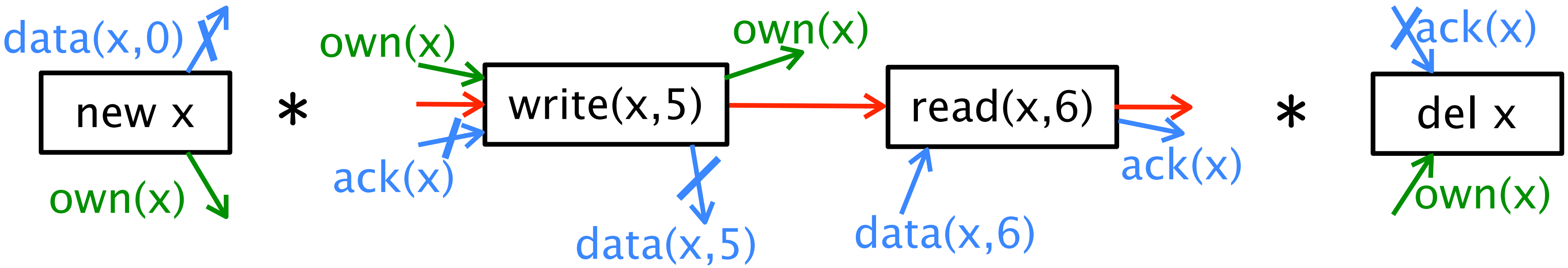


=



Sequential composition

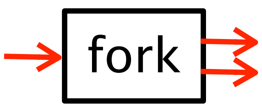
- ⟦var x in {x:=5; assume x=6}⟧ =



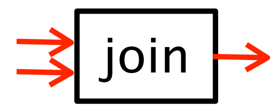
∩ varconstraints(x)

Parallel composition

- $[\![C_1 || C_2]\!] =$



$*_{seq} \; ([\![C1]\!] \; *_{par} \; [\![C2]\!]) \; *_{seq}$



where $t_1 \circ_{par} t_2$ is only defined when:

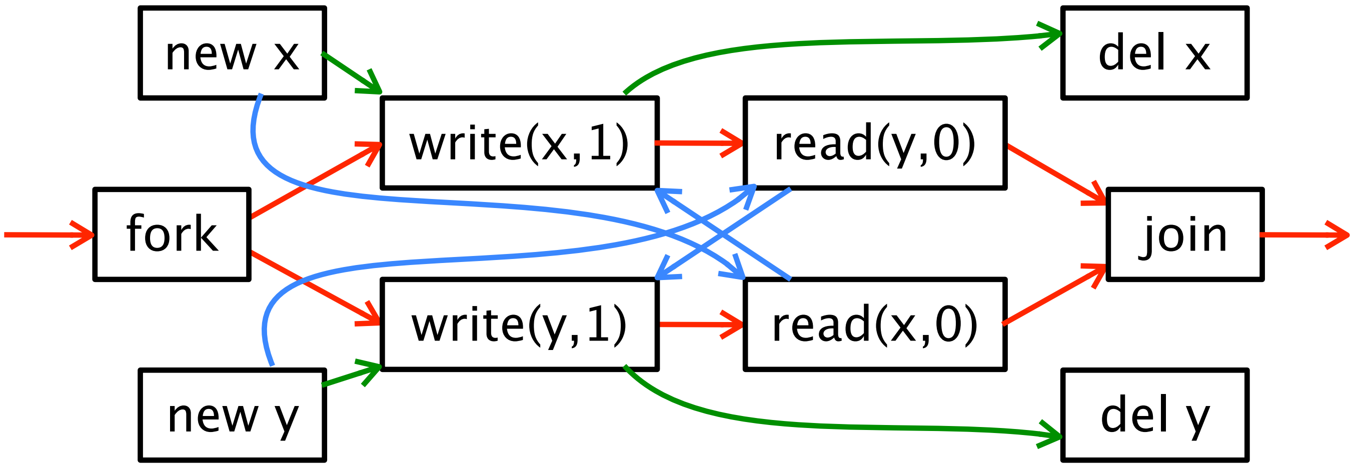
$$\mathrm{danglingCtrl}(t_1) \cap \mathrm{danglingCtrl}(t_2) = \emptyset$$

and $*_{par}$ is the lifted version of $\circ_{par}$

Weak memory

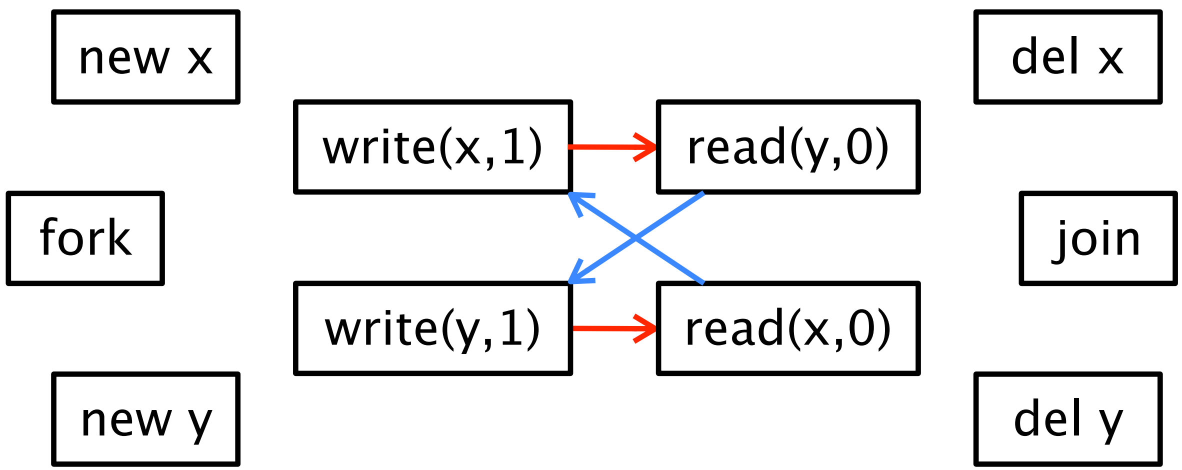
Weak memory

```
var x in var y in {
  write(x,1); || write(y,1);
  read(y,0)   || read(x,0)
}
```

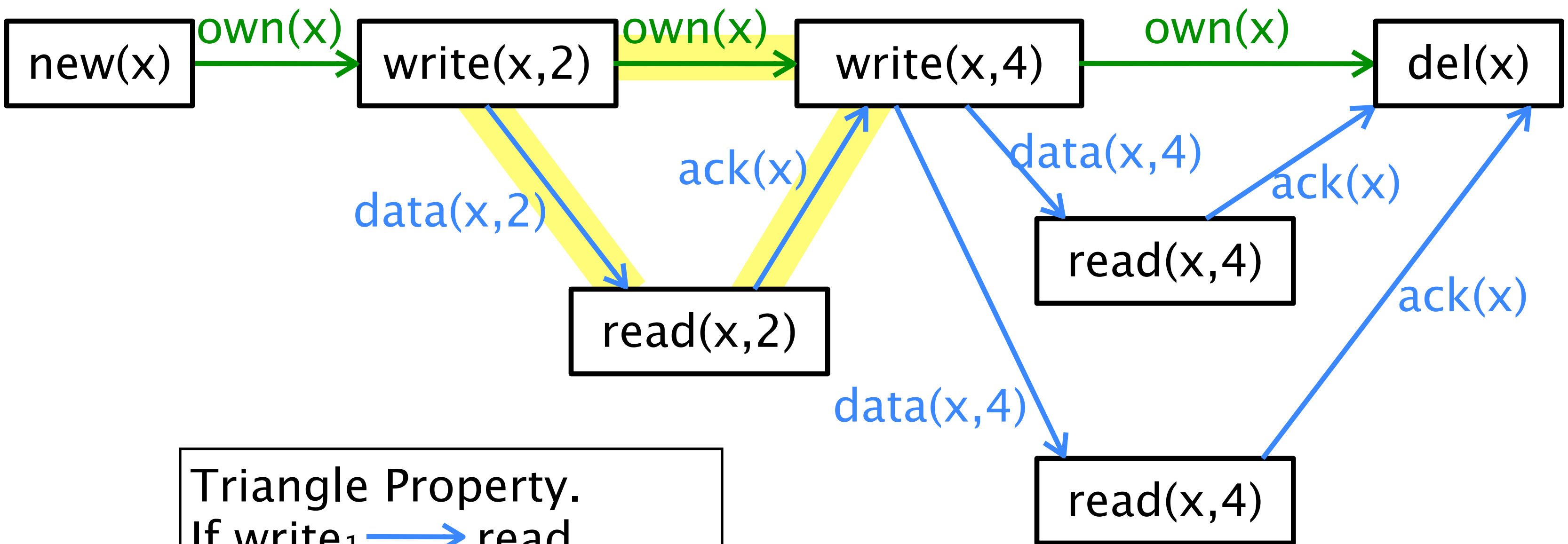


Weak memory

```
var x in var y in {
  write(x,1); || write(y,1);
  read(y,0)   || read(x,0)
}
```

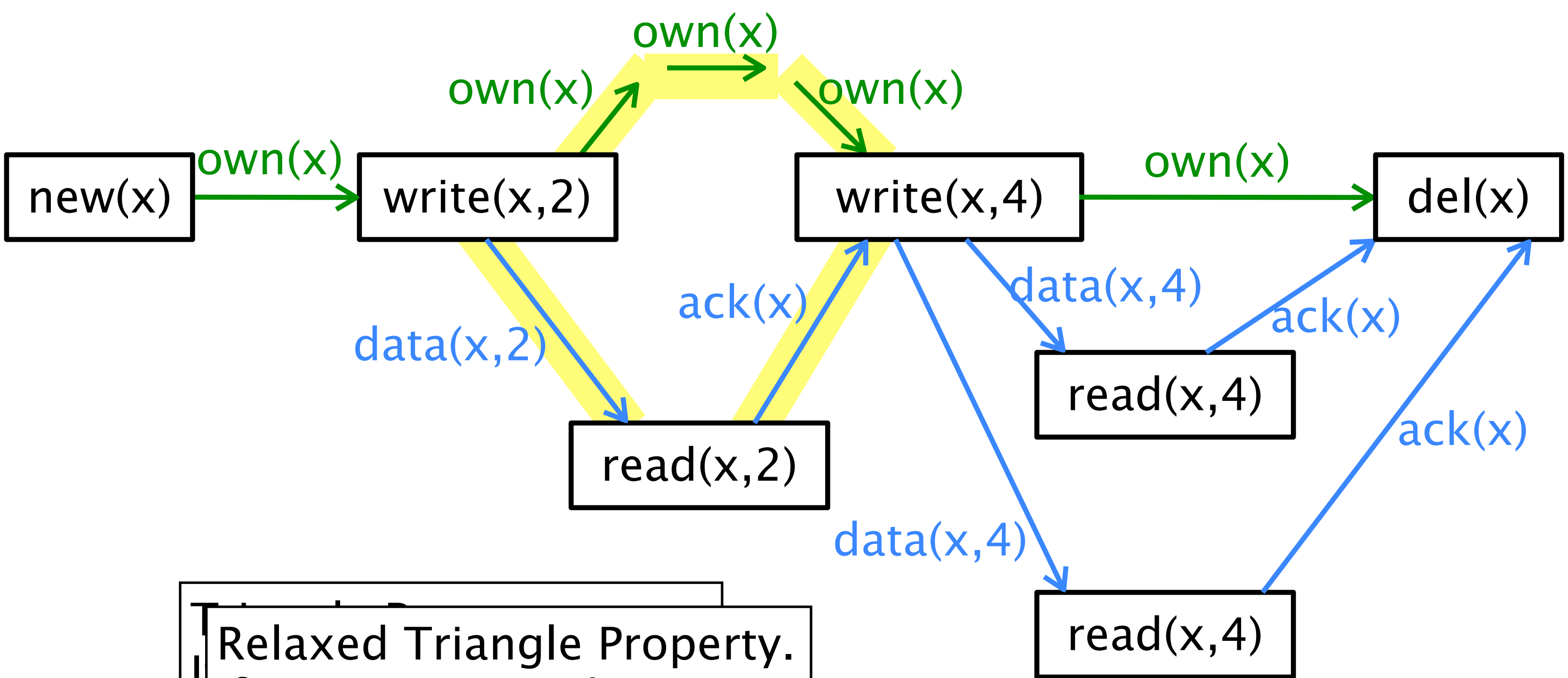


Variables



Triangle Property.
If $write_1$ → read
and read → $write_2$
then $write_1$ → $write_2$

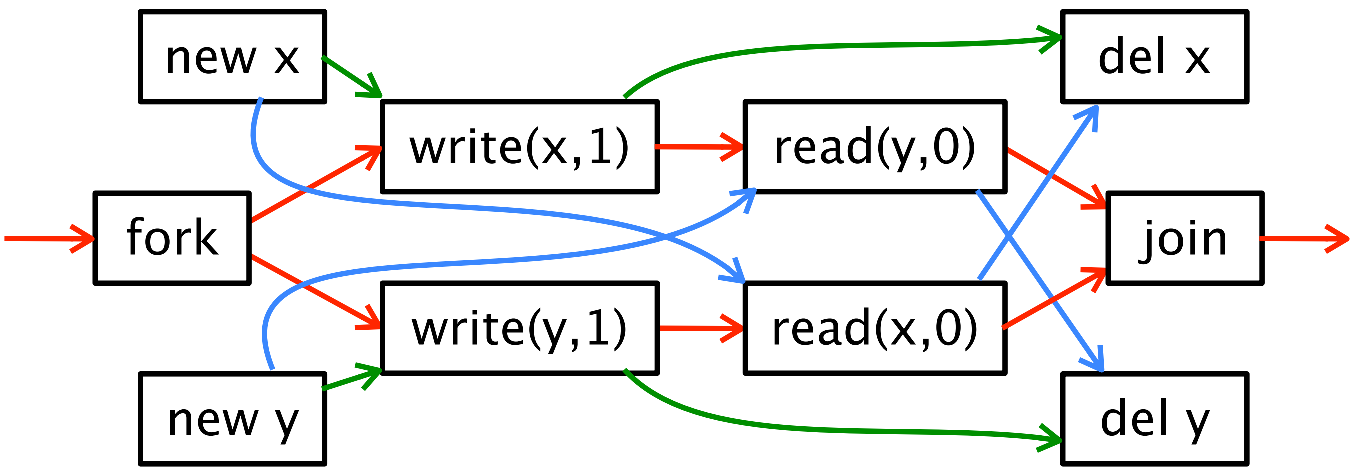
Variables



Relaxed Triangle Property.
If $write_1 \rightarrow read$
and $read \rightarrow write_2$
then $write_1 \rightarrow^{+} write_2$

Weak memory

```
var x in var y in {
  write(x,1);  ||  write(y,1);
  read(y,0)    ||  read(x,0)
}
```



Summary

- A model of concurrency, communication and weak memory, based on dataflow
- Next steps:
  - automate the generation of traces?
  - use as a basis for a program logic for weak memory?

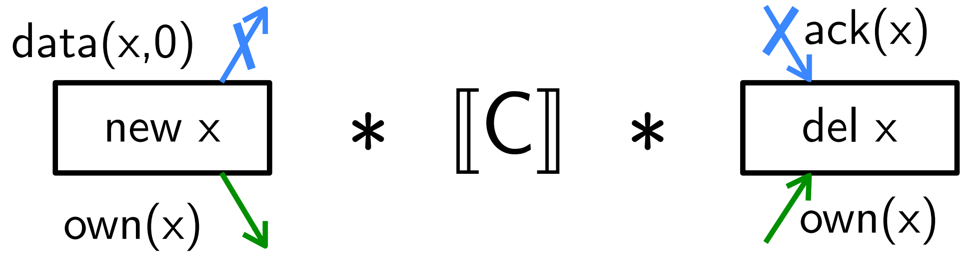
Spare slides

Use of separation logic laws

- We can use laws of separation logic to prove theorems about our model, such as commutativity of local variable declarations

Use of separation logic laws

- ⟦var x in C⟧ =



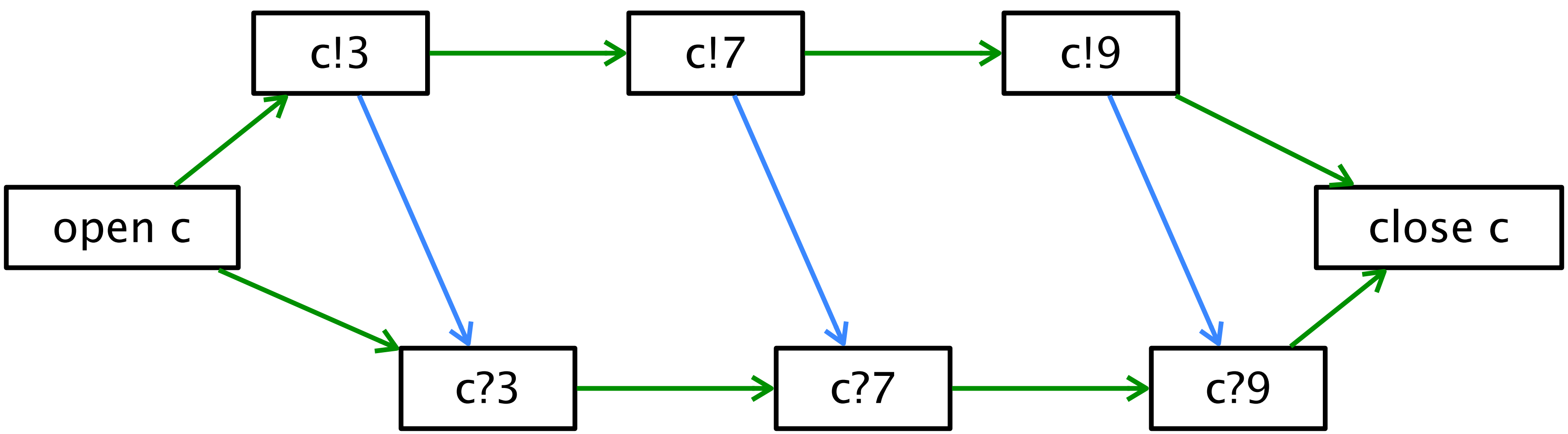
∩ varconstraints(x)

Use of separation logic laws

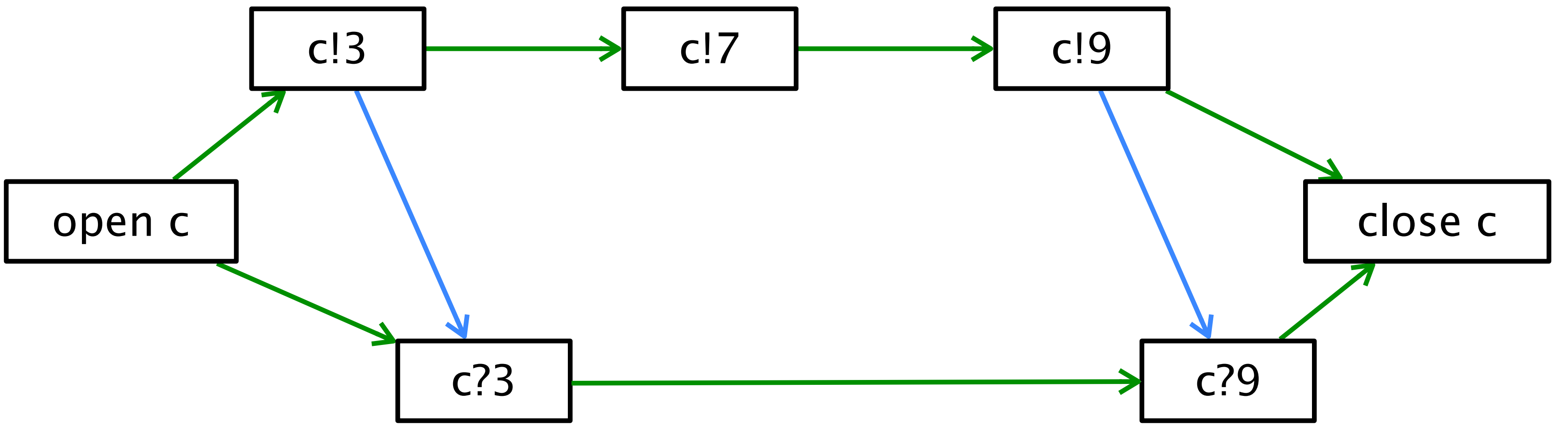
- $[\![\text{var x in C}]\!] = ([\![C]\!] * \text{nd}_x) \cap v_x$
- $[\![\text{var y in var x in C}]\!] = [\![\text{var x in var y in C}]\!]$?
- $((([\![C]\!] * \text{nd}_x) \cap v_x) * \text{nd}_y) \cap v_y$
  $= ([\![C]\!] * \text{nd}_x * \text{nd}_y) \cap v_x \cap v_y$
- $(P \wedge Q) * R \;=\; P{*}R \wedge Q{*}R$
  (provided R is precise)

Communication

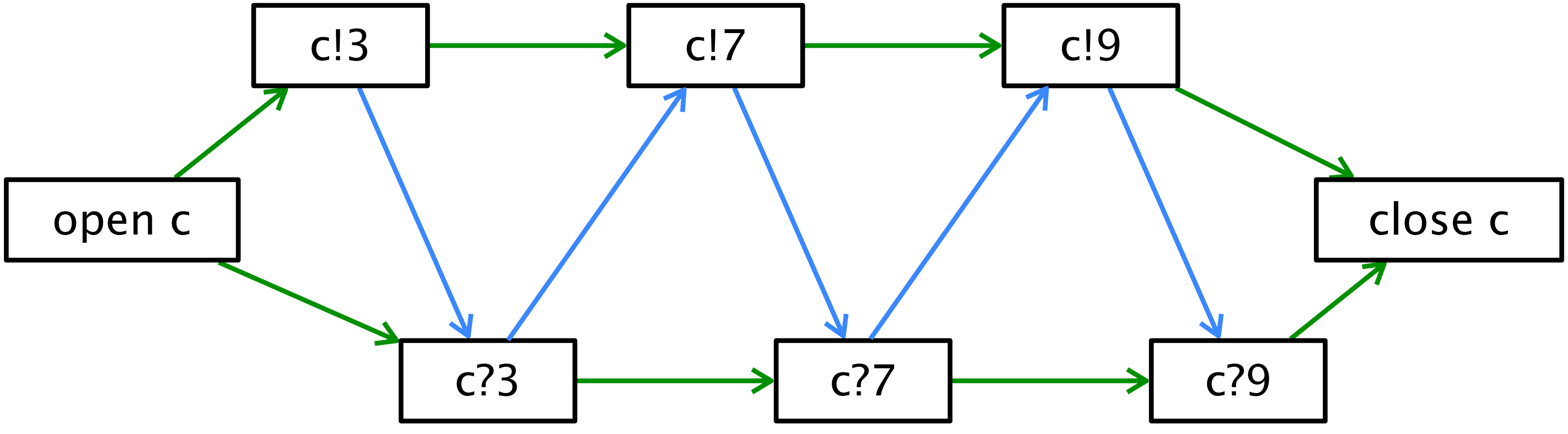
Well-behaved channel



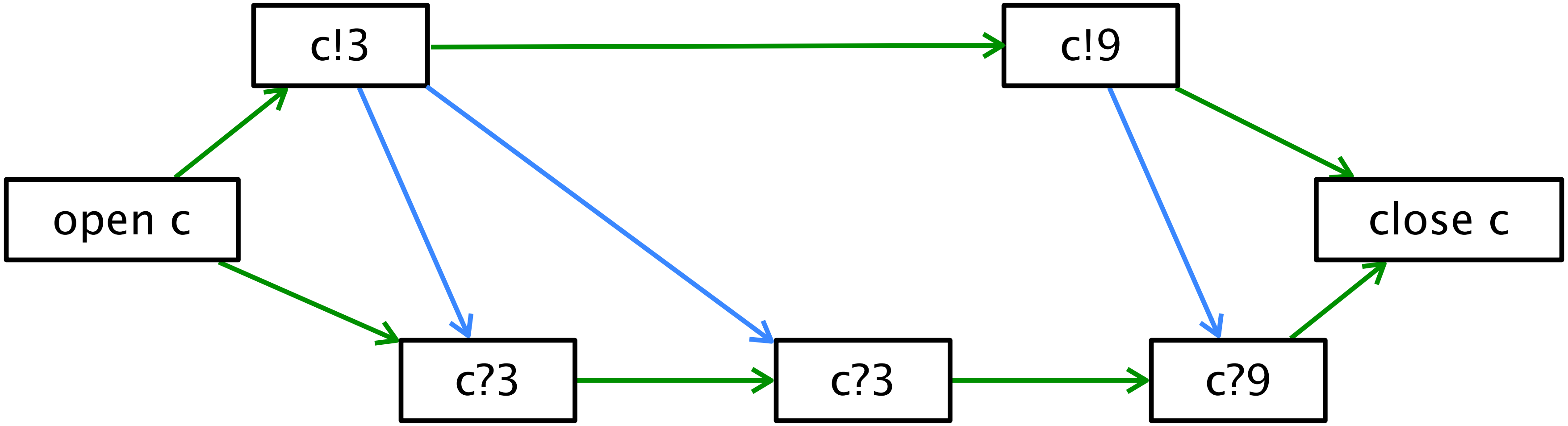
Lossy channel



Singly-buffered channel



Stuttering channel



Re-ordering channel