A dataflow model of concurrency, communication and weak memory

John Wickerson & Tony Hoare

Microsoft Research

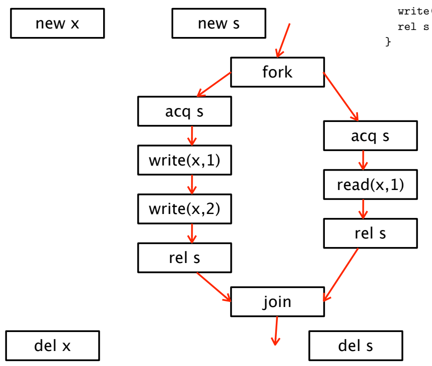
Cambridge Concurrency Workshop
5–6 July 2010

Example

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

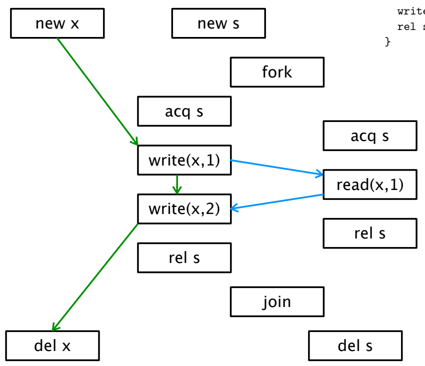

Example

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



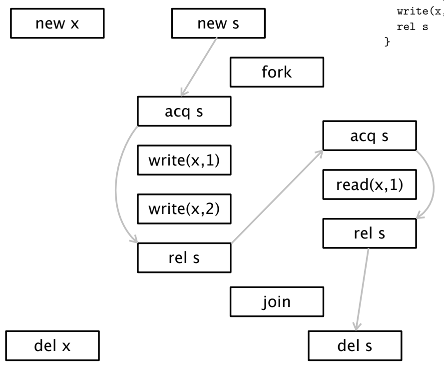
Example

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



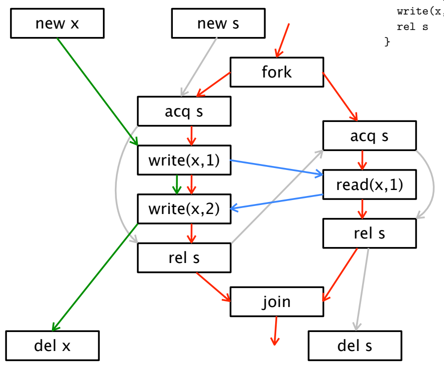
Example

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



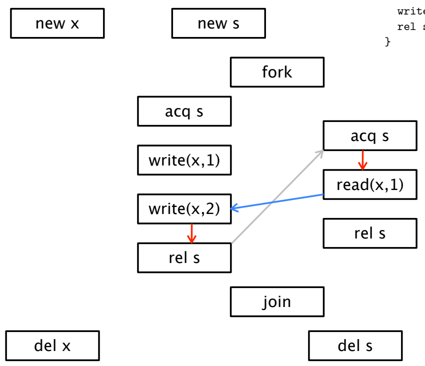
Example

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



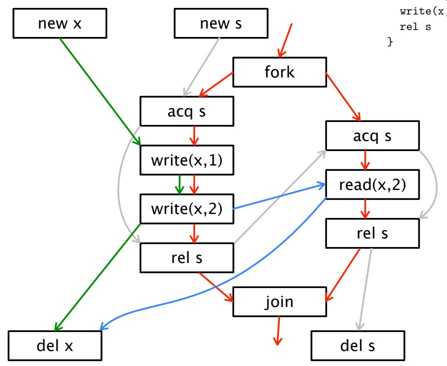
Example

```
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

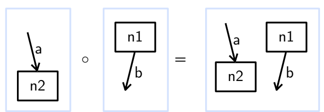
- We model a program as a set of possible traces
- We separate various kinds of flow
  - data flow, control flow, ownership transfer
- Our model is stateless
  - good for modelling weak memory and asynchronous communication

Traces

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

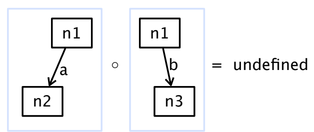
Traces

- A composition operator:



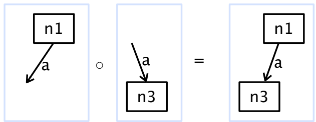
Traces

- A composition operator:



Traces

- A composition operator:



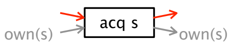
- Lifted to sets of traces:
  T∗U = {t∘u | t∈T, u∈U}

A denotational semantics

⟦-⟧ : Command → $\mathbb{P}_{fin}$(Trace)

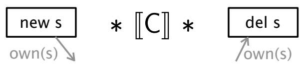
Locks

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

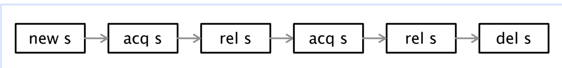


- ⟦rel s⟧ =

- ⟦lock s in C⟧ =

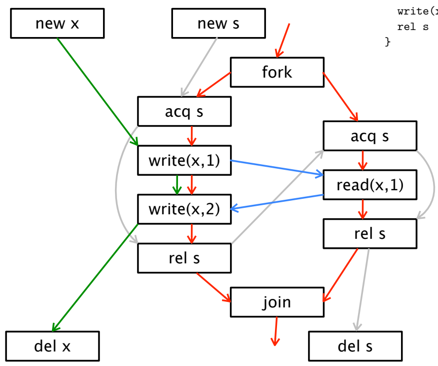


∩ lockconstraints(s)



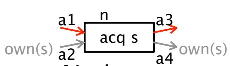
Example

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



Locks

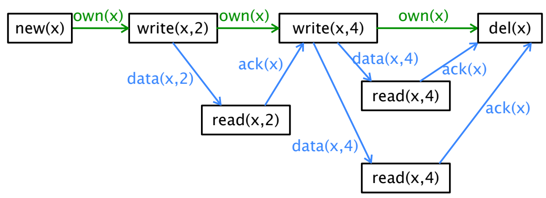
- ⟦acq s⟧ = {



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

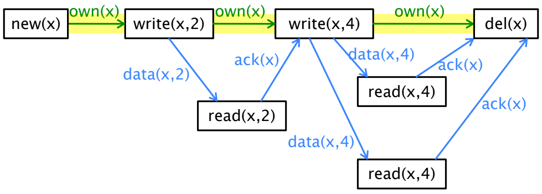
Variables

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



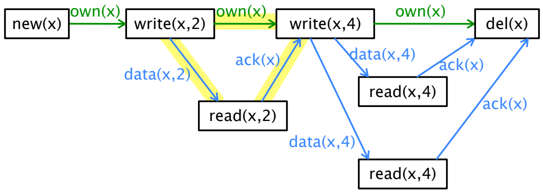
Variables

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



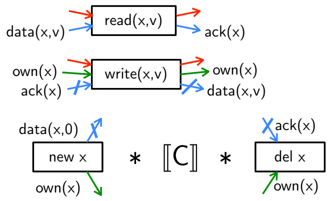
Variables

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



Variables

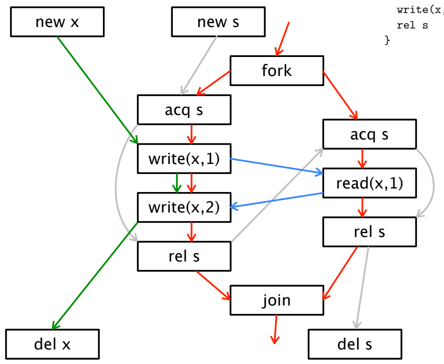
- C ::= … | var x in C | write(x,v) | read(x,v)
- ⟦read(x,v)⟧ =
- ⟦write(x,v)⟧ =
- ⟦var x in C⟧ =



∩ varconstraints(x)

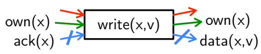
Example

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

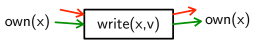


Variables

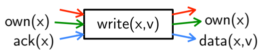
- ⟦write(x,v)⟧ =



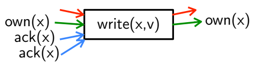
=



∪



∪



∪ …

Assignments and assumptions

- $⟦x := f(y_1, …, y_n)⟧$ =
  $∪\{⟦read(y_1,v_1); … ; read(y_n,v_n); write(x,v)⟧$
  $| f(v_1,…,v_n) = v \}$

- $⟦assume\ p(x_1, …, x_n)⟧$ =
  $∪\{⟦read(x_1,v_1); … ; read(x_n,v_n)⟧$
  $| p(v_1,…,v_n) = true \}$

Sequential composition

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

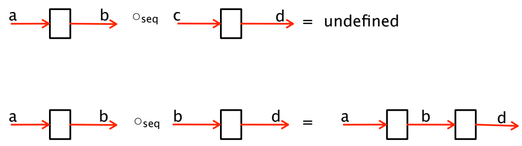
  where $t_1 \circ_{seq} t_2$ is only defined when:

  $$outCtrl(t_1) = inCtrl(t_2)$$

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

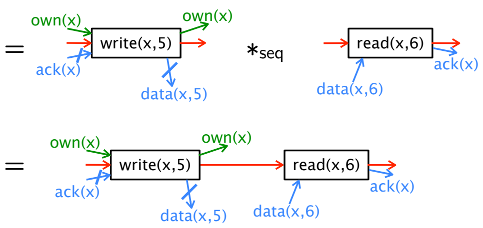
Sequential composition

- Examples:



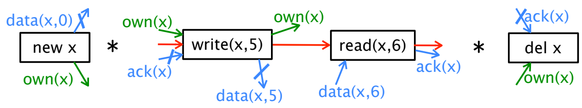
Sequential composition

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



Sequential composition

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



$\cap$ varconstraints(x)

Parallel composition

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

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

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

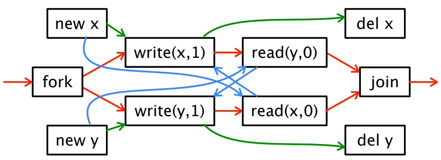
  $$danglingCtrl(t_1) \cap danglingCtrl(t_2) = \varnothing$$

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

Weak memory

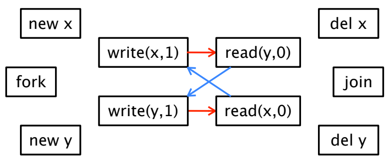
Weak memory

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

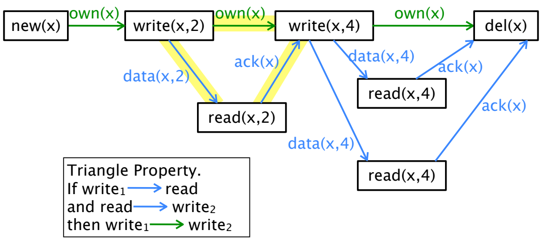


Weak memory

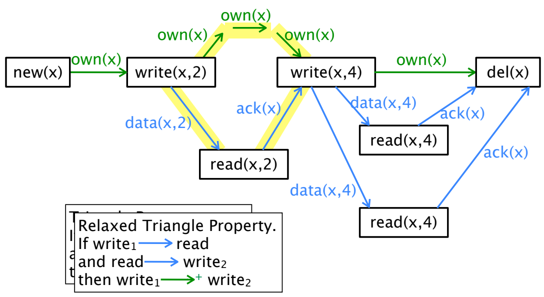
```
var x in var y in {
  write(x,1); || write(y,1);
  read(y,0)   || read(x,0)
}
```



Variables

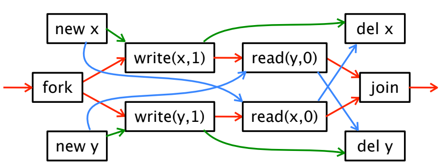


Variables



Weak memory

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



Summary

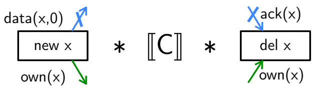
- A model of concurrency, communication and weak memory, based on dataflow
- Next steps:
  - automate the generation of traces?
  - use as a basis for a program logic for weak memory?

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

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

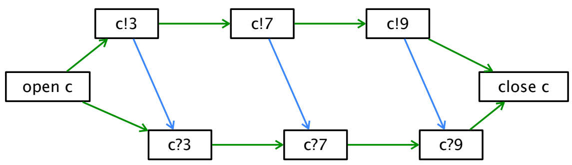
∩ varconstraints(x)

Use of separation logic laws

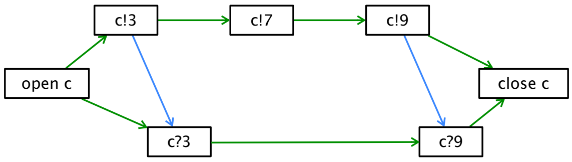
- $[\![\text{var x in C}]\!] = ([\![C]\!] * nd_x) \cap v_x$
- $[\![\text{var y in var x in C}]\!] = [\![\text{var x in var y in C}]\!]$?
- $((([\![C]\!] * nd_x) \cap v_x) * nd_y) \cap v_y$
  $= ([\![C]\!] * nd_x * nd_y) \cap v_x \cap v_y$
- $(P \wedge Q) * R \;=\; P{*}R \wedge Q{*}R$
  (provided R is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

Communication

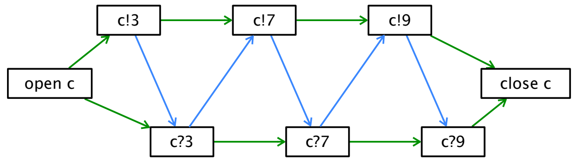
Well-behaved channel



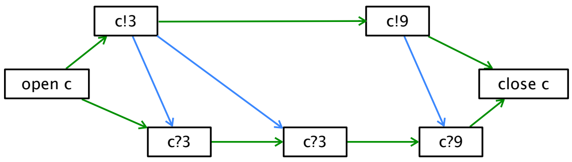
Lossy channel



Singly-buffered channel



Stuttering channel



Re-ordering channel