A dataflow model of concurrency, communication and weak memory

John Wickerson & Tony Hoare
Example

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

acq s;
read(x,1);
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;
  write(x,1);
  read(x,1);
  write(x,2);
  rel s
  rel s
}
```
Example

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

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

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

new x

new s

fork

acq s

write(x,1)

write(x,2)

rel s

fork

acq s

read(x,1)

rel s

join

del x

del s
Example

```
lock s in var x in {
  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, \( N \in \mathcal{P}_{\text{fin}} \text{Node} \)
  - ArrowSet, \( A \in \mathcal{P}_{\text{fin}} \text{Arrow} \)
  - Labelling, \( L \in N \rightarrow \text{Label} \)
  - Valuation, \( V \in A \rightarrow \text{Value} \)
  - HeadMap, \( H \in A \rightarrow N \)
  - TailMap, \( T \in A \rightarrow N \)
Traces

• A composition operator:
Traces

- A composition operator:

\[
\begin{array}{c}
\text{n1} \\
\downarrow a \\
\text{n2}
\end{array} \circ \\
\begin{array}{c}
\text{n1} \\
\downarrow b \\
\text{n3}
\end{array} = \text{undefined}
\]
Traces

• A composition operator:

\[ \text{n1} \xrightarrow{a} \text{n3} \quad \otimes \quad \text{n3} = \text{n1} \]

• Lifted to sets of traces:

\[ T \ast U = \{ t \circ u \mid t \in T, u \in U \} \]
A denotational semantics

\[[\cdot]\]: \text{Command} \rightarrow \mathcal{P}_{\text{fin}}(\text{Trace})
• $C ::= \ldots \mid \text{lock } s \text{ in } C \mid \text{acq } s \mid \text{rel } s$

• $\llbracket \text{acq } s \rrbracket =$

• $\llbracket \text{rel } s \rrbracket =$

• $\llbracket \text{lock } s \text{ in } C \rrbracket =$

$\llbracket \text{lock } s \text{ in } C \rrbracket =$

$\mathbf{new } s \ast \llbracket C \rrbracket \ast \mathbf{del } s$

$n \text{ lockconstraints}(s)$
Example

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

fork

acq s

write(x,1)

write(x,2)

rel s

join

del x

del s
Locks

- $[\text{acq } s] = \{ n \in \text{Node}, a_1, a_2, a_3, a_4 \in \text{Arrow}, a_1, a_2, a_3, a_4 \text{ all distinct} \}$
Variables

- $C ::= \ldots \mid \text{var } x \text{ in } C \mid \text{write}(x,v) \mid \text{read}(x,v)$
Variables

- $C ::= \ldots \mid \text{var } x \text{ in } C \mid \text{write}(x,v) \mid \text{read}(x,v)$
Variables

- $C ::= \ldots \mid \text{var } x \text{ in } C \mid \text{write}(x,v) \mid \text{read}(x,v)$
Variables

- \( C ::= \ldots \mid \text{var } x \text{ in } C \mid \text{write}(x,v) \mid \text{read}(x,v) \)
- \( [\text{read}(x,v)] = \)
- \( [\text{write}(x,v)] = \)
- \( [\text{var } x \text{ in } C] = \)

\[ \text{data}(x,v) \quad \text{ack}(x) \]
\[ \text{own}(x) \quad \text{data}(x,v) \quad \text{ack}(x) \]
\[ \text{new } x \quad \text{del } x \]
\[ \cap \text{var constraints}(x) \]
Example

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

new x
new s
fork
acq s
write(x,1)
write(x,2)
rel s
fork
acq s
read(x,1)
rel s
join
del s
del x
Variables

- $\llbracket \text{write}(x,v) \rrbracket = \text{own}(x) \quad \text{write}(x,v) \quad \text{own}(x)\]
  \[\text{ack}(x) \quad \text{data}(x,v)\]

\[
\begin{align*}
\quad = \quad \text{own}(x) \quad \text{write}(x,v) \quad \text{own}(x) \\
\quad \quad \quad \quad \quad \quad \text{ack}(x) \quad \text{data}(x,v) \\
\quad U \quad \text{own}(x) \quad \text{write}(x,v) \quad \text{own}(x) \\
\quad U \quad \text{ack}(x) \quad \text{data}(x,v) \\
\quad U \quad \text{ack}(x) \quad \text{ack}(x) \quad \text{own}(x) \\
\quad U \quad \ldots
\end{align*}
\]
Assignments and assumptions

- $\llbracket x := f(y_1, \ldots, y_n) \rrbracket =$
  $\bigcup \{ \llbracket \text{read}(y_1,v_1); \ldots; \text{read}(y_n,v_n); \text{write}(x,v) \rrbracket$
  $\mid f(v_1,\ldots,v_n) = v \}$

- $\llbracket \text{assume } p(x_1, \ldots, x_n) \rrbracket =$
  $\bigcup \{ \llbracket \text{read}(x_1,v_1); \ldots; \text{read}(x_n,v_n) \rrbracket$
  $\mid p(v_1,\ldots,v_n) = \text{true} \}$
Sequential composition

\[ [C_1;C_2] = [C_1] \ast_{\text{seq}} [C_2] \]

where \( t_1 \circ_{\text{seq}} t_2 \) is only defined when:

\[ \text{outCtrl}(t_1) = \text{inCtrl}(t_2) \]

and \( \ast_{\text{seq}} \) is the lifted version of \( \circ_{\text{seq}} \).
Sequential composition

• Examples:

\[ \begin{array}{c}
a \rightarrow \quad b \quad \circ_{\text{seq}} \quad c \rightarrow \quad d = \text{undefined} \\
\end{array}\]

\[ \begin{array}{c}
a \rightarrow \quad b \quad \circ_{\text{seq}} \quad b \rightarrow \quad d = \quad a \rightarrow \quad b \rightarrow \quad d \\
\end{array}\]
Sequential composition

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

\[ \left[ \text{var } x \text{ in } \{x:=5; \text{ assume } x=6\} \right] = \]

\[ \star \text{ new } x \rightarrow \text{ own}(x) \rightarrow \text{ write}(x,5) \rightarrow \text{ own}(x) \rightarrow \text{ read}(x,6) \rightarrow \text{ own}(x) \rightarrow \star \text{ del } x \]

∩ \text{ var constraints}(x)
Parallel composition

• $[[C_1||C_2]] = f_{\text{fork}} \xrightarrow{*_{\text{seq}}} ([[C_1] *_{\text{par}} [C_2]]) *_{\text{seq}} f_{\text{join}}$

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

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

and $*_{\text{par}}$ is the lifted version of $\circ_{\text{par}}$
Weak memory
var x in var y in {
    write(x, 1); \parallel write(y, 1);
    read(y, 0) \parallel read(x, 0)
}
Weak memory

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

```
fork
```

```
new x
```

```
new y
```

```
write(x,1)  ---
```

```
write(y,1)  ||
```

```
read(y,0)   ---
```

```
read(x,0)
```

```
del x
```

```
del y
```

```
join
```
Triangle Property.
If write$_1$ → read
and read → write$_2$
then write$_1$ → write$_2$
Variables

Triangle Property.
If \( \text{write}_1 \rightarrow \text{read} \) and \( \text{read} \rightarrow \text{write}_2 \) then \( \text{write}_1 \rightarrow^+ \text{write}_2 \)

Relaxed Triangle Property.
If \( \text{write}_1 \rightarrow \text{read} \) and \( \text{read} \rightarrow \text{write}_2 \) then \( \text{write}_1 \rightarrow^+ \text{write}_2 \)
Weak memory

```plaintext
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

• \[ [\text{var } x \text{ in } C] = \begin{array}{c}
  \text{new } x \\
  \text{data}(x,0) \\
  \text{own}(x)
\end{array} * \begin{array}{c}
  \text{C} \\
  \text{del } x \\
  \text{ack}(x)
\end{array} * \begin{array}{c}
  \text{var constraints}(x)
\end{array} \]
Use of separation logic laws

- $\left[\text{var } x \text{ in } C\right] = (\left[\text{C}\right] \ast \text{nd}_x) \cap v_x$
- $\left[\text{var } y \text{ in } \text{var } x \text{ in } C\right] = \left[\text{var } x \text{ in } \text{var } y \text{ in } C\right]$ ?
- $((\left[\text{C}\right] \ast \text{nd}_x) \cap v_x) \ast \text{nd}_y) \cap v_y$
  \[= (\left[\text{C}\right] \ast \text{nd}_x \ast \text{nd}_y) \cap v_x \cap v_y\]
- $(P \land Q) \ast R = P \ast R \land Q \ast R$
  \[(\text{provided } R \text{ is precise})\]
Communication
Well-behaved channel

open c

c!3 → c!7 → c!9

c?3 → c?7 → c?9

close c
Lossy channel

open c

c!3 -> c!7 -> c!9

c?3

c?9

close c
Singly-buffered channel

open c

c?3

c!3

open c

c?7

c!7

open c

c?9

c!9

close c

c?9
Stuttering channel

open c

c!3

c?3

c?3

c!9

c?9

close c
Re-ordering channel

open c

c!3

c!7

c!9

c?7

c?3

c?9

close c