Dialectical refinement: Rescuing programming from the logicians

Richard Bornat
Professor of Computer Programming
School of Engineering and Information Sciences
Middlesex University

5th November 2008

There are <u>two</u> programming problems

There are <u>two</u> programming problems

- The novice programming problem

There are two programming problems

- The novice programming problem
- The expert programming problem

There are <u>two</u> programming problems

- The novice programming problem
- The expert programming problem

Both are interesting.

There are two programming problems

- The novice programming problem
- The expert programming problem

Both are interesting. This talk is about the second problem.

There are <u>two</u> programming problems

- The novice programming problem
- The expert programming problem

Both are interesting. This talk is about the second problem.

Most programming 'languages', *especially* including Java, address the expert programming problem.

Programming as a creative act

Programming as a creative act

I take ‘programming’ to be the whole business: conception, design, construction, verification, testing …

Programming as a creative act

I take ‘programming’ to be the whole business: conception, design, construction, verification, testing ...

I’m very interested in ‘correct’ programs: ones with specification and verification.

Programming as a creative act

I take ‘programming’ to be the whole business: conception, design, construction, verification, testing . . .

I’m very interested in ‘correct’ programs: ones with specification and verification.

And I’m interested in *inventing* correct programs.

Programming as a creative act

I take ‘programming’ to be the whole business: conception, design, construction, verification, testing . . .

I’m very interested in ‘correct’ programs: ones with specification and verification.

And I’m interested in *inventing* correct programs.

(I claim that) invention always involves trial and error.

Programming as a creative act

I take 'programming' to be the whole business: conception, design, construction, verification, testing . . .

I'm very interested in 'correct' programs: ones with specification and verification.

And I'm interested in *inventing* correct programs.

(I claim that) invention always involves trial and error.

Can trial and error be 'logical'? Or is it illogical guesswork?

Two ways to program

Two ways to program

I can invent a program, in a blinding flash of creativity

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .)

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .). Logicians call this *ad-hoc programming*.

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming.

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? . . .

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? . . .

Logicians would like us to start with the specification

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? . . .

Logicians would like us to start with the specification; write a magical (unimplementable) program which satisfies the specification

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? . . .

Logicians would like us to start with the specification; write a magical (unimplementable) program which satisfies the specification; then *refine* the magical program step-by-step towards something that can be used

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? . . .

Logicians would like us to start with the specification; write a magical (unimplementable) program which satisfies the specification; then *refine* the magical program step-by-step towards something that can be used; using refinement steps which preserve the property that the program satisfies the specification.

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it ...). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? ...

Logicians would like us to start with the specification; write a magical (unimplementable) program which satisfies the specification; then *refine* the magical program step-by-step towards something that can be used; using refinement steps which preserve the property that the program satisfies the specification.

The logicians' way of programming produces a result that is *correct by construction*.

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it ...). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? ...

Logicians would like us to start with the specification; write a magical (unimplementable) program which satisfies the specification; then *refine* the magical program step-by-step towards something that can be used; using refinement steps which preserve the property that the program satisfies the specification.

The logicians' way of programming produces a result that is *correct by construction*.

Ad-hoc programs are rarely specified, even more rarely verified.

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it ...). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? ...

Logicians would like us to start with the specification; write a magical (unimplementable) program which satisfies the specification; then *refine* the magical program step-by-step towards something that can be used; using refinement steps which preserve the property that the program satisfies the specification.

The logicians' way of programming produces a result that is *correct by construction*.

Ad-hoc programs are rarely specified, even more rarely verified.

Can we defend program invention?

Two ways to program

I can invent a program, in a blinding flash of creativity; and then verify it (*of course* I wouldn't test it . . .). Logicians call this *ad-hoc programming*.

Before the verification I have to think up a logical specification. I don't need the specification to start programming. Or do I? . . .

Logicians would like us to start with the specification; write a magical (unimplementable) program which satisfies the specification; then *refine* the magical program step-by-step towards something that can be used; using refinement steps which preserve the property that the program satisfies the specification.

The logicians' way of programming produces a result that is *correct by construction*.

Ad-hoc programs are rarely specified, even more rarely verified.

Can we defend program invention? Can we give it a nicer name?

Dialectical programming

Dialectical programming

Lakatos, in “Proofs and Refutations”, shows the evolution over time of Euler’s conjecture about polyhedral solids $Vertices - Edges + Faces = 2$ (e.g for a cube $V = 8$, $E = 12$, $F = 6$).

Dialectical programming

Lakatos, in “Proofs and Refutations”, shows the evolution over time of Euler’s conjecture about polyhedral solids
$Vertices - Edges + Faces = 2$ (e.g for a cube $V = 8$, $E = 12$, $F = 6$).

The *conjecture* leads to a *proof*

Dialectical programming

Lakatos, in "Proofs and Refutations", shows the evolution over time of Euler's conjecture about polyhedral solids
$Vertices - Edges + Faces = 2$ (e.g for a cube $V = 8$, $E = 12$, $F = 6$).

The *conjecture* leads to a *proof* which is challenged by a *counter-example* (e.g. a polyhedron with a tunnel, a polyhedron with a central cavity)

Dialectical programming

Lakatos, in “Proofs and Refutations”, shows the evolution over time of Euler’s conjecture about polyhedral solids
$Vertices - Edges + Faces = 2$ (e.g for a cube $V = 8$, $E = 12$, $F = 6$).

The *conjecture* leads to a *proof* which is challenged by a *counter-example* (e.g. a polyhedron with a tunnel, a polyhedron with a central cavity) which leads to a refinement of the conjecture, a new proof, a deeper understanding.

Dialectical programming

Lakatos, in "Proofs and Refutations", shows the evolution over time of Euler's conjecture about polyhedral solids
$\mathit{Vertices} - \mathit{Edges} + \mathit{Faces} = 2$ (e.g for a cube $V = 8$, $E = 12$, $F = 6$).

The *conjecture* leads to a *proof* which is challenged by a *counter-example* (e.g. a polyhedron with a tunnel, a polyhedron with a central cavity) which leads to a refinement of the conjecture, a new proof, a deeper understanding.

This is intentionally reminiscent of Hegel's dialectic "Thesis plus antithesis yields synthesis".

Dialectical programming

Lakatos, in "Proofs and Refutations", shows the evolution over time of Euler's conjecture about polyhedral solids
$Vertices - Edges + Faces = 2$ (e.g for a cube $V = 8$, $E = 12$, $F = 6$).

The *conjecture* leads to a *proof* which is challenged by a *counter-example* (e.g. a polyhedron with a tunnel, a polyhedron with a central cavity) which leads to a refinement of the conjecture, a new proof, a deeper understanding.

This is intentionally reminiscent of Hegel's dialectic "Thesis plus antithesis yields synthesis". (But we are simple people, we won't bother with Hegel.)

Dialectical programming

Lakatos, in “Proofs and Refutations”, shows the evolution over time of Euler’s conjecture about polyhedral solids
$Vertices - Edges + Faces = 2$ (e.g for a cube $V = 8$, $E = 12$, $F = 6$).

The *conjecture* leads to a *proof* which is challenged by a *counter-example* (e.g. a polyhedron with a tunnel, a polyhedron with a central cavity) which leads to a refinement of the conjecture, a new proof, a deeper understanding.

This is intentionally reminiscent of Hegel’s dialectic “Thesis plus antithesis yields synthesis”. (But we are simple people, we won’t bother with Hegel.)

Can the dialectic reach places that logical refinement cannot?

Dialectical programming

Lakatos, in “Proofs and Refutations”, shows the evolution over time of Euler’s conjecture about polyhedral solids $\mathit{Vertices} - \mathit{Edges} + \mathit{Faces} = 2$ (e.g for a cube $V = 8$, $E = 12$, $F = 6$).

The *conjecture* leads to a *proof* which is challenged by a *counter-example* (e.g. a polyhedron with a tunnel, a polyhedron with a central cavity) which leads to a refinement of the conjecture, a new proof, a deeper understanding.

This is intentionally reminiscent of Hegel’s dialectic “Thesis plus antithesis yields synthesis”. (But we are simple people, we won’t bother with Hegel.)

Can the dialectic reach places that logical refinement cannot? I think so.

The single-place buffer

The single-place buffer

In an aeroplane you have *sensors* – e.g. temperature, pressure, airflow – and you have *monitors* – e.g. black-box recorders, autopilots, display screens.

The single-place buffer

In an aeroplane you have *sensors* – e.g. temperature, pressure, airflow – and you have *monitors* – e.g. black-box recorders, autopilots, display screens.

We want a *simple* interface between a monitor and a sensor.

The single-place buffer

In an aeroplane you have *sensors* – e.g. temperature, pressure, airflow – and you have *monitors* – e.g. black-box recorders, autopilots, display screens.

We want a *simple* interface between a monitor and a sensor.

The simplest is a single-place buffer: the sensor writes to it, the monitor reads.

The single-place buffer

In an aeroplane you have *sensors* – e.g. temperature, pressure, airflow – and you have *monitors* – e.g. black-box recorders, autopilots, display screens.

We want a *simple* interface between a monitor and a sensor.

The simplest is a single-place buffer: the sensor writes to it, the monitor reads.

The sensor may write faster than the monitor reads (so there may be missed values).

The single-place buffer

In an aeroplane you have *sensors* – e.g. temperature, pressure, airflow – and you have *monitors* – e.g. black-box recorders, autopilots, display screens.

We want a *simple* interface between a monitor and a sensor.

The simplest is a single-place buffer: the sensor writes to it, the monitor reads.

The sensor may write faster than the monitor reads (so there may be missed values).

The monitor may read faster than the sensor writes (so there may be repeated values).

The single-place buffer

In an aeroplane you have *sensors* – e.g. temperature, pressure, airflow – and you have *monitors* – e.g. black-box recorders, autopilots, display screens.

We want a *simple* interface between a monitor and a sensor.

The simplest is a single-place buffer: the sensor writes to it, the monitor reads.

The sensor may write faster than the monitor reads (so there may be missed values).

The monitor may read faster than the sensor writes (so there may be repeated values).

But the monitor *must* always get complete values, not half-written values, not half of one value and half of the next.

How it can go wrong

How it can go wrong

The sensor is a clock, displaying minutes m and seconds s in two shared integer variables.

How it can go wrong

The sensor is a clock, displaying minutes m and seconds s in two shared integer variables.

Each second it does

$$\textsf{if } s = 59 \textsf{ then } s := 0; m := m + 1 \textsf{ else } s := s + 1 \textsf{ fi}$$

How it can go wrong

The sensor is a clock, displaying minutes m and seconds s in two shared integer variables.

Each second it does

$$\textsf{if } s = 59 \textsf{ then } s := 0; m := m + 1 \textsf{ else } s := s + 1 \textsf{ fi}$$

Between $s := 0$ and $m := m + 1$ the clock is slow (by 60 seconds). If the monitor reads at that instant it sees a half-written value.

How it can go wrong

The sensor is a clock, displaying minutes m and seconds s in two shared integer variables.

Each second it does

$$\textsf{if } s = 59 \textsf{ then } s := 0; m := m + 1 \textsf{ else } s := s + 1 \textsf{ fi}$$

Between $s := 0$ and $m := m + 1$ the clock is slow (by 60 seconds). If the monitor reads at that instant it sees a half-written value.

If the sensor does $m := m + 1; s := 0$ then the clock can be seen as fast (by 59 seconds).

Dijkstra's solution: atomic accesses

Dijkstra's solution: atomic accesses

$$
\text{local } b = \mathit{null} \text{ in}
\left(
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \langle b := w \rangle \\
\\
\\
\end{array}
\;\middle\|\;
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \text{local } y \text{ in} \\
\qquad\qquad \langle y := b \rangle; \\
\qquad\qquad \text{return } y \\
\qquad\quad \text{ni}
\end{array}
\right)
$$

Dijkstra's solution: atomic accesses

$$
\text{local } b = \mathit{null} \text{ in}
$$

$$
\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \langle b := w \rangle & \mathrm{read}(\,) \mathrel{\hat{=}} \text{local } y \text{ in} \\
& \qquad\qquad \langle y := b \rangle; \\
& \qquad\qquad \text{return } y \\
& \qquad\quad \text{ni}
\end{array}\right)
$$

Atomicity means indivisibility. Interleaving will do.

Dijkstra's solution: atomic accesses

$$
\begin{array}{l}
\textsf{local}\ b = \mathit{null}\ \textsf{in} \\
\left(
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \langle b := w \rangle \\
\\
\\
\\
\end{array}
\;\middle\|\;
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \textsf{local}\ y\ \textsf{in} \\
\qquad\qquad \langle y := b \rangle; \\
\qquad\qquad \textsf{return}\ y \\
\qquad\quad \textsf{ni}
\end{array}
\right)
\end{array}
$$

Atomicity means indivisibility. Interleaving will do.

Atomicity can be done with 'semaphores' as on the railways (block signalling).

Dijkstra's solution: atomic accesses

$$\text{local } b = null, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l}
\quad \ldots & \quad \ldots \\
\quad \mathrm{write}(w) \mathrel{\hat=} & \quad \mathrm{read}(\,) \mathrel{\hat=} \\
\qquad \langle b := w; ws := ws.w\rangle & \qquad \text{local } y \text{ in} \\
 & \qquad\quad \langle y := b; rs := rs.y\rangle; \\
 & \qquad\quad \text{return } y \\
 & \qquad \text{ni}
\end{array}\right)$$

Atomicity means indivisibility. Interleaving will do.

Atomicity can be done with 'semaphores' as on the railways (block signalling).

For specification purposes I add a couple of auxiliary variables ws and rs.

Dijkstra's solution: atomic accesses

$$\text{local } b = null, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \langle b := w; ws := ws.w\rangle & \quad \text{local } y \text{ in} \\
 & \qquad \langle y := b; rs := rs.y\rangle; \\
 & \qquad \text{return } y \\
 & \quad \text{ni}
\end{array}\right)$$

Atomicity means indivisibility. Interleaving will do.

Atomicity can be done with 'semaphores' as on the railways (block signalling).

For specification purposes I add a couple of auxiliary variables ws and rs.

The specification is that the read sequence destuttered is a subsequence of ws and the last element of ws is always in b.
$\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b$.

Dijkstra's solution: atomic accesses

$$\text{local } b = null, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \langle b := w; ws := ws.w \rangle & \quad \text{local } y \text{ in} \\
 & \qquad \langle y := b; rs := rs.y \rangle; \\
 & \qquad \text{return } y \\
 & \quad \text{ni}
\end{array}\right)$$

Atomicity means indivisibility. Interleaving will do.

Atomicity can be done with 'semaphores' as on the railways (block signalling).

For specification purposes I add a couple of auxiliary variables ws and rs.

The specification is that the read sequence destuttered is a subsequence of ws and the last element of ws is always in b.
$\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b$.

But atomicity means waiting, and waiting isn't *simple* or even *certain*.

Cut down waiting with a two-slot buffer

Cut down waiting with a two-slot buffer

$$\text{local } b = null, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l} \begin{array}{l} \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} \\ \quad \langle b := w; ws := ws.w \rangle \end{array} & \begin{array}{l} \ldots \\ \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \text{local } y \text{ in} \\ \qquad \langle y := b; rs := rs.y \rangle; \\ \qquad \text{return } y \\ \quad \text{ni} \end{array} \end{array}\right)$$

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l} \begin{array}{l} \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} \\ \quad \left\langle \begin{array}{l} c[!l] := w; l := !l; \\ ws := ws.w \end{array} \right\rangle \end{array} & \begin{array}{l} \ldots \\ \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \text{local } y \text{ in} \\ \qquad \langle y := c[l]; rs := rs.y \rangle; \\ \qquad \text{return } y \\ \quad \text{ni} \end{array} \end{array}\right)$$

Cut down waiting with a two-slot buffer

$$
\text{local } b = null, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
\left(
\begin{array}{l}
\ldots \\
\text{write}(w) \mathrel{\hat{=}} \\
\quad \langle b := w; ws := ws.w \rangle
\end{array}
\;\middle\|\;
\begin{array}{l}
\ldots \\
\text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y \text{ in} \\
\quad\quad \langle y := b; rs := rs.y \rangle; \\
\quad\quad \text{return } y \\
\quad \text{ni}
\end{array}
\right)
$$

$$
\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
\left(
\begin{array}{l}
\ldots \\
\text{write}(w) \mathrel{\hat{=}} \\
\quad \left\langle \begin{array}{l} c[!l] := w; l := !l; \\ ws := ws.w \end{array} \right\rangle
\end{array}
\;\middle\|\;
\begin{array}{l}
\ldots \\
\text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y \text{ in} \\
\quad\quad \langle y := c[l]; rs := rs.y \rangle; \\
\quad\quad \text{return } y \\
\quad \text{ni}
\end{array}
\right)
$$

Data refinement!

Cut down waiting with a two-slot buffer

$$\text{local } b = null, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l} \begin{array}{l} \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} \\ \quad \langle b := w; ws := ws.w \rangle \end{array} & \begin{array}{l} \ldots \\ \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \text{local } y \text{ in} \\ \quad\quad \langle y := b; rs := rs.y \rangle; \\ \quad\quad \text{return } y \\ \quad \text{ni} \end{array} \end{array}\right)$$

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l} \begin{array}{l} \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} \\ \quad \left\langle \begin{array}{l} c[!l] := w; l := !l; \\ ws := ws.w \end{array} \right\rangle \end{array} & \begin{array}{l} \ldots \\ \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \text{local } y \text{ in} \\ \quad\quad \langle y := c[l]; rs := rs.y \rangle; \\ \quad\quad \text{return } y \\ \quad \text{ni} \end{array} \end{array}\right)$$

Data refinement! Now $\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l]$.

Simplify atomicity in the writer

$$\mathsf{local}\ c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle\ \mathsf{in}$$

$$\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \left\langle \begin{array}{l} c[!l] := w; l := !l; \\ ws := ws.w \end{array} \right\rangle \\
\\
\\
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \mathsf{local}\ y\ \mathsf{in} \\
\qquad \langle y := c[l]; rs := rs.y \rangle; \\
\qquad \mathsf{return}\ y \\
\quad \mathsf{ni}
\end{array}
\end{array}\right)$$

Simplify atomicity in the writer

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \left\langle \begin{array}{l} c[!l] := w; l := !l; \\ ws := ws.w \end{array} \right\rangle \\
\\
\\
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y \text{ in} \\
\qquad \langle y := c[l]; rs := rs.y \rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}\right)$$

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\langle wt := !l \rangle\rangle; \\
\qquad \langle c[wt] := w \rangle; \\
\qquad \langle\langle l := wt; ws := ws.w \rangle\rangle \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y \text{ in} \\
\qquad \langle y := c[l]; rs := rs.y \rangle; \\
\qquad \text{return } y \\
\quad \text{ni} \\
\\
\end{array}
\end{array}\right)$$

Simplify atomicity in the writer

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \left\langle \begin{array}{l} c[!l] := w; l := !l; \\ ws := ws.w \end{array} \right\rangle & \quad \text{local } y \text{ in} \\
 & \qquad \langle y := c[l]; rs := rs.y \rangle; \\
 & \qquad \text{return } y \\
 & \quad \text{ni}
\end{array}\right)$$

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle y := c[l]; rs := rs.y \rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \text{return } y \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \quad \text{ni} \\
\quad \text{ni} &
\end{array}\right)$$

Stores naturally serialise small reads and writes.

Simplify atomicity in the writer

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l} \ldots & \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \left\langle \begin{array}{l} c[!l] := w; l := !l; \\ ws := ws.w \end{array} \right\rangle & \quad \text{local } y \text{ in} \\ & \qquad \langle y := c[l]; rs := rs.y \rangle; \\ & \qquad \text{return } y \\ & \quad \text{ni} \end{array}\right)$$

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l} \ldots & \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \text{local } wt \text{ in} & \quad \text{local } y \text{ in} \\ \qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle y := c[l]; rs := rs.y \rangle; \\ \qquad \langle c[wt] := w \rangle; & \qquad \text{return } y \\ \qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \quad \text{ni} \\ \quad \text{ni} & \end{array}\right)$$

Stores naturally serialise small reads and writes. Still $\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l]$.

(Can't) simplify atomicity in the reader

(Can't) simplify atomicity in the reader

$$
\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
$$

$$
\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle y := c[l]; rs := rs.y \rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \text{return } y \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \quad \text{ni} \\
\quad \text{ni} &
\end{array}\right)
$$

(Can't) simplify atomicity in the reader

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle y := c[l]; rs := rs.y \rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \text{return } y \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \quad \text{ni} \\
\quad \text{ni} &
\end{array}\right)$$

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \langle y := c[rt]; rs := rs.y \rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \qquad \text{return } y \\
\quad \text{ni} & \quad \text{ni}
\end{array}\right)$$

(Can't) simplify atomicity in the reader

$$\mathsf{local}\ c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle\ \mathsf{in}$$
$$\left(\begin{array}{l} \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} \\ \quad \mathsf{local}\ wt\ \mathsf{in} \\ \qquad \langle\!\langle wt := !l \rangle\!\rangle; \\ \qquad \langle c[wt] := w \rangle; \\ \qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \\ \quad \mathsf{ni} \end{array} \;\middle\|\; \begin{array}{l} \ldots \\ \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \mathsf{local}\ y\ \mathsf{in} \\ \qquad \langle y := c[l]; rs := rs.y \rangle; \\ \qquad \mathsf{return}\ y \\ \quad \mathsf{ni} \end{array}\right)$$

$$\mathsf{local}\ c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle\ \mathsf{in}$$
$$\left(\begin{array}{l} \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} \\ \quad \mathsf{local}\ wt\ \mathsf{in} \\ \qquad \langle\!\langle wt := !l \rangle\!\rangle; \\ \qquad \langle c[wt] := w \rangle; \\ \qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \\ \quad \mathsf{ni} \end{array} \;\middle\|\; \begin{array}{l} \ldots \\ \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \mathsf{local}\ y, rt\ \mathsf{in} \\ \qquad \langle\!\langle rt := l \rangle\!\rangle; \\ \qquad \langle y := c[rt]; rs := rs.y \rangle; \\ \qquad \mathsf{return}\ y \\ \quad \mathsf{ni} \end{array}\right)$$

Looks plausible, but it's broken.

(Can't) simplify atomicity in the reader

$$\text{local } c[2] = (null, null), l = 0, \color{red}{ws} = \langle\,\rangle.null, \color{red}{rs} = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle y := c[l]; \color{red}{rs} := \color{red}{rs}.y \rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \text{return } y \\
\qquad \langle\!\langle l := wt; \color{red}{ws} := \color{red}{ws}.w \rangle\!\rangle & \quad \text{ni} \\
\quad \text{ni} &
\end{array}\right)$$

$$\text{local } c[2] = (null, null), l = 0, \color{red}{ws} = \langle\,\rangle.null, \color{red}{rs} = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \langle y := c[rt]; \color{red}{rs} := \color{red}{rs}.y \rangle; \\
\qquad \langle\!\langle l := wt; \color{red}{ws} := \color{red}{ws}.w \rangle\!\rangle & \qquad \text{return } y \\
\quad \text{ni} & \quad \text{ni}
\end{array}\right)$$

Looks plausible, but it's broken. Still $\color{red}{ws}_\Omega = c[l]$

(Can't) simplify atomicity in the reader

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y \text{ in} \\
\qquad \langle y := c[l]; rs := rs.y \rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}\right)$$

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle y := c[rt]; rs := rs.y \rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}\right)$$

Looks plausible, but it's broken. Still $ws_\Omega = c[l]$, but no longer $\lfloor rs \rfloor \preccurlyeq ws$.

What goes wrong?

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

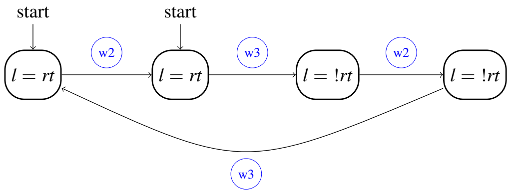
$$\left(\begin{array}{l||l}
\quad \ldots & \quad \ldots \\
\quad \mathrm{write}(w) \mathrel{\hat{=}} & \quad \mathrm{read}(\,) \mathrel{\hat{=}} \\
\qquad \text{local } wt \text{ in} & \qquad \text{local } y, rt \text{ in} \\
\qquad\quad \langle\!\langle wt := !l \rangle\!\rangle; \quad \text{w1} & \qquad\quad \langle\!\langle rt := l \rangle\!\rangle; \quad \text{r1} \\
\qquad\quad \langle c[wt] := w \rangle; \quad \text{w2} & \qquad\quad \langle y := c[rt]; rs := rs.y \rangle; \quad \text{r2} \\
\qquad\quad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \quad \text{w3} & \qquad\quad \text{return } y \\
\qquad \text{ni} & \qquad \text{ni}
\end{array}\right)$$

What goes wrong?

local $c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle$ in

$$
\left(
\begin{array}{l l || l l}
\ldots & & \ldots & \\
\mathrm{write}(w) \mathrel{\hat{=}} & & \mathrm{read}(\,) \mathrel{\hat{=}} & \\
\quad \textsf{local } wt \textsf{ in} & & \quad \textsf{local } y, rt \textsf{ in} & \\
\qquad \langle\langle wt := !l \rangle\rangle; & \text{w1} & \qquad \langle\langle rt := l \rangle\rangle; & \text{r1} \\
\qquad \langle c[wt] := w \rangle; & \text{w2} & \qquad \langle y := c[rt]; rs := rs.y \rangle; & \text{r2} \\
\qquad \langle\langle l := wt; ws := ws.w \rangle\rangle & \text{w3} & \qquad \textsf{return } y & \\
\quad \textsf{ni} & & \quad \textsf{ni} &
\end{array}
\right)
$$

From the point of view of the reader, after $\langle\langle rt := l \rangle\rangle$, the writer behaves like this finite-state machine:

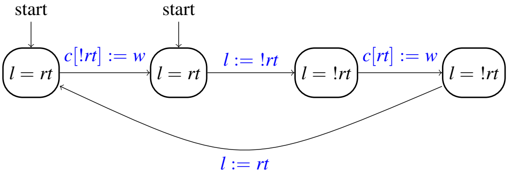


What goes wrong?

local $c[2] = (null, null), l = 0,$ $ws = \langle\,\rangle.null,$ $rs = \langle\,\rangle$ in

$$
\left(
\begin{array}{l|l}
\begin{array}{ll}
\ldots & \\
\text{write}(w) \mathrel{\hat{=}} & \\
\quad \text{local } wt \text{ in} & \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \text{w1} \\
\qquad \langle c[wt] := w \rangle; & \text{w2} \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \text{w3} \\
\quad \text{ni} &
\end{array}
&
\begin{array}{ll}
\ldots & \\
\text{read}(\,) \mathrel{\hat{=}} & \\
\quad \text{local } y, rt \text{ in} & \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; & \text{r1} \\
\qquad \langle y := c[rt]; rs := rs.y \rangle; & \text{r2} \\
\qquad \text{return } y & \\
\quad \text{ni} &
\end{array}
\end{array}
\right)
$$

From the point of view of the reader, after $\langle\!\langle rt := l \rangle\!\rangle$, the writer behaves like this finite-state machine:



What goes wrong?

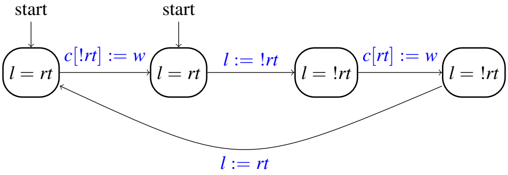
local $c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle$ in

$\ldots$
write$(w)\ \hat{=}$
  local wt in
    $\langle\!\langle wt := !l \rangle\!\rangle;$ w1
    $\langle c[wt] := w \rangle;$ w2
    $\langle\!\langle l := wt;\ ws := ws.w \rangle\!\rangle$ w3
  ni

$\|$

$\ldots$
read$(\,)\ \hat{=}$
  local y, rt in
    $\langle\!\langle rt := l \rangle\!\rangle;$ r1
    $\langle y := c[rt];\ rs := rs.y \rangle;$ r2
    return y
  ni

From the point of view of the reader, after $\langle\!\langle rt := l \rangle\!\rangle$, the writer behaves like this finite-state machine:



If the reader comes in at box 1 or 2 and reads at box 4, it will see the second value written

What goes wrong?

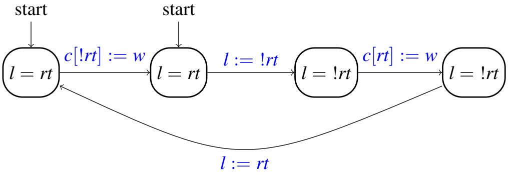
local $c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle$ in

$\ldots$
write$(w) \mathrel{\hat{=}}$
  local wt in
    $\langle\!\langle wt := !l \rangle\!\rangle$; w1
    $\langle c[wt] := w \rangle$; w2
    $\langle\!\langle l := wt; ws := ws.w \rangle\!\rangle$ w3
  ni

$\|$

$\ldots$
read$(\,) \mathrel{\hat{=}}$
  local y, rt in
    $\langle\!\langle rt := l \rangle\!\rangle$; r1
    $\langle y := c[rt]; rs := rs.y \rangle$; r2
    return y
  ni

From the point of view of the reader, after $\langle\!\langle rt := l \rangle\!\rangle$, the writer behaves like this finite-state machine:



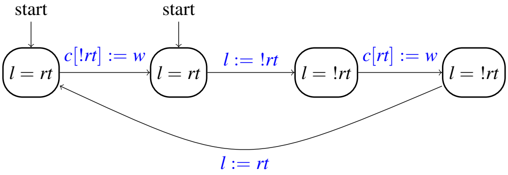
If the reader comes in at box 1 or 2 and reads at box 4, it will see the second value written; if it then comes back quickly, it can see the first thing written!!

What goes wrong?

local $c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle$ in

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \quad \text{w1} & \qquad \langle\!\langle rt := l \rangle\!\rangle; \quad \text{r1} \\
\qquad \langle c[wt] := w \rangle; \quad \text{w2} & \qquad \langle y := c[rt]; rs := rs.y \rangle; \quad \text{r2} \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \quad \text{w3} & \qquad \text{return } y \\
\quad \text{ni} & \quad \text{ni}
\end{array}\right)$$

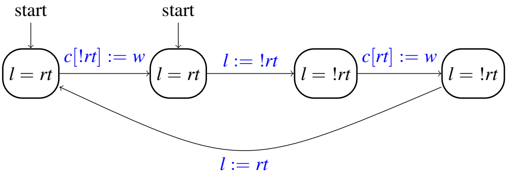
From the point of view of the reader, after $\langle\!\langle rt := l \rangle\!\rangle$, the writer behaves like this finite-state machine:



If the reader comes in at box 1 or 2 and reads at box 4, it will see the second value written; if it then comes back quickly, it can see the first thing written!! Also note $\langle c[rt] := w \rangle \parallel \langle y := c[rt] \rangle$.

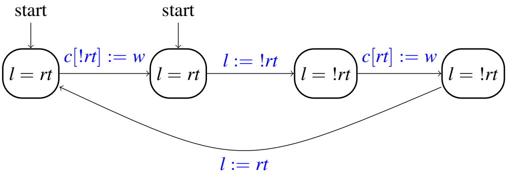
Can we repair it (1)?

All our problems (ordering, collisions) are caused by the third action



Can we repair it (1)?

All our problems (ordering, collisions) are caused by the third action



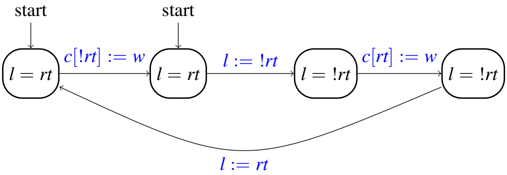
Can we detect when that action happens?

$$
\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
$$

$$
\left(\begin{array}{l|l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \langle y := c[rt]; rs := rs.y \rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \qquad \text{return } y \\
\quad \text{ni} & \quad \text{ni}
\end{array}\right)
$$

Can we repair it (1)?

All our problems (ordering, collisions) are caused by the third action



Can we detect when that action happens?

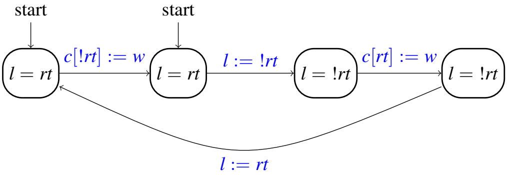
$$\textsf{local}\ c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle\ \textsf{in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat=} & \mathrm{read}(\,) \mathrel{\hat=} \\
\quad \textsf{local}\ wt\ \textsf{in} & \quad \textsf{local}\ y, rt\ \textsf{in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \langle y := c[rt]; rs := rs.y \rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \qquad \textsf{return}\ y \\
\quad \textsf{ni} & \quad \textsf{ni}
\end{array}\right)$$

No, because the writer can't tell the difference between the first and the third actions.

Can we repair it (2)?

All our problems (ordering, collisions) are caused by the third action

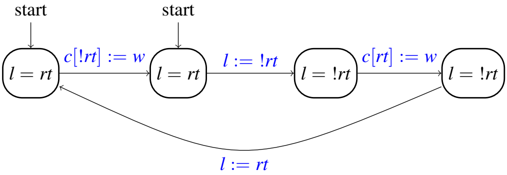


Can we detect when it *might* happen?

$$
\begin{array}{l}
\textsf{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \textsf{ in} \\
\left(\begin{array}{l|l}
\quad \ldots & \quad \ldots \\
\quad \mathrm{write}(w) \mathrel{\hat=} & \quad \mathrm{read}(\,) \mathrel{\hat=} \\
\quad\quad \textsf{local } wt \textsf{ in} & \quad\quad \textsf{local } y, rt \textsf{ in} \\
\quad\quad\quad \langle\!\langle wt := !l \rangle\!\rangle; & \quad\quad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\quad\quad\quad \langle c[wt] := w \rangle; & \quad\quad\quad \langle y := c[rt]; rs := rs.y \rangle; \\
\quad\quad\quad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \quad\quad\quad \textsf{return } y \\
\quad\quad \textsf{ni} & \quad\quad \textsf{ni}
\end{array}\right)
\end{array}
$$

Can we repair it (2)?

All our problems (ordering, collisions) are caused by the third action



Can we detect when it *might* happen?

local $c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle$ in

$$\left(\begin{array}{l||l}
\quad \ldots & \quad \ldots \\
\quad \mathrm{write}(w) \mathrel{\hat{=}} & \quad \mathrm{read}(\,) \mathrel{\hat{=}} \\
\qquad \textsf{local } wt \textsf{ in} & \qquad \textsf{local } y, rt \textsf{ in} \\
\qquad\quad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad\quad \langle c[wt] := w \rangle; & \qquad\quad \langle y := c[rt]; rs := rs.y \rangle; \\
\qquad\quad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle & \qquad\quad \textsf{return } y \\
\qquad \textsf{ni} & \qquad \textsf{ni}
\end{array}\right)$$

Yes: it becomes possible after the second action.

A repair (back to correctness)

The writer signals when disaster becomes possible; the reader incorporates the signal in its answer.

$$\text{local } c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\text{write}(w) \mathrel{\hat=} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \text{false} \rangle\!\rangle; \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\text{read}(\,) \mathrel{\hat=} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle ok := \text{true} \rangle\!\rangle; \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle y := c[rt] \rangle; \\
\qquad \langle\!\langle rt = ok; rs := rs.(rt, y) \rangle\!\rangle; \\
\qquad \text{return } (rt, y) \\
\quad \text{ni}
\end{array}
\end{array}\right)$$

A repair (back to correctness)

The writer signals when disaster becomes possible; the reader incorporates the signal in its answer.

$$\mathsf{local}\ c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle\ \mathsf{in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \mathsf{local}\ wt\ \mathsf{in} & \quad \mathsf{local}\ y, rt\ \mathsf{in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle ok := \text{true} \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \qquad \langle y := c[rt] \rangle; \\
\qquad \langle\!\langle ok := \text{false} \rangle\!\rangle; & \qquad \langle\!\langle rt = ok; rs := rs.(rt, y) \rangle\!\rangle; \\
\quad \mathsf{ni} & \qquad \mathsf{return}\ (rt, y) \\
 & \quad \mathsf{ni}
\end{array}\right)$$

– and then we notice that we don't need atomic buffer accesses any more.

A repair (back to correctness)

The writer signals when disaster becomes possible; the reader incorporates the signal in its answer.

$$\mathsf{local}\ c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle\ \mathsf{in}$$

$$\left(\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \mathsf{local}\ wt\ \mathsf{in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle; \\
\quad \mathsf{ni}
\end{array}
\;\Bigg\|\;
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \mathsf{local}\ y, rt\ \mathsf{in} \\
\qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad y := c[rt]; \\
\qquad \langle\!\langle rt = ok; rs := rs.(rt, y) \rangle\!\rangle; \\
\qquad \mathsf{return}\ (rt, y) \\
\quad \mathsf{ni}
\end{array}\right)$$

– and then we notice that we don't need atomic buffer accesses any more.

A repair (back to correctness)

The writer signals when disaster becomes possible; the reader incorporates the signal in its answer.

$$\text{local } c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l}
\quad \ldots & \quad \ldots \\
\quad \text{write}(w) \mathrel{\hat{=}} & \quad \text{read}(\,) \mathrel{\hat{=}} \\
\quad\quad \text{local } wt \text{ in} & \quad\quad \text{local } y, rt \text{ in} \\
\quad\quad\quad \langle\!\langle wt := !l \rangle\!\rangle; & \quad\quad\quad \langle\!\langle ok := \text{true} \rangle\!\rangle; \\
\quad\quad\quad c[wt] := w; & \quad\quad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\quad\quad\quad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \quad\quad\quad y := c[rt]; \\
\quad\quad\quad \langle\!\langle ok := \text{false} \rangle\!\rangle; & \quad\quad\quad \langle\!\langle rt = ok; rs := rs.(rt, y) \rangle\!\rangle; \\
\quad\quad \text{ni} & \quad\quad\quad \text{return } (rt, y) \\
 & \quad\quad \text{ni}
\end{array}\right)$$

– and then we notice that we don’t need atomic buffer accesses any more.

If $\widetilde{rs}$ is rs with the $(\text{false}, \_)$ results taken out and the true labels discarded, then we have $\lfloor \widetilde{rs} \rfloor \preccurlyeq ws \wedge ws_{\Omega} = c[l]$.

Summary so far, and a criticism

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free)

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free) to an atomic double-slot buffer (ditto)

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free) to an atomic double-slot buffer (ditto) to a faulty not so completely atomic double-slot buffer (still not wait-free)

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free) to an atomic double-slot buffer (ditto) to a faulty not so completely atomic double-slot buffer (still not wait-free) to a working wait-free non-atomic double-slot buffer that tells us when it's succeeded.

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free) to an atomic double-slot buffer (ditto) to a faulty not so completely atomic double-slot buffer (still not wait-free) to a working wait-free non-atomic double-slot buffer that tells us when it's succeeded.

We certainly haven't proceeded by "a sequence of true understatements" (Lakatos);

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free) to an atomic double-slot buffer (ditto) to a faulty not so completely atomic double-slot buffer (still not wait-free) to a working wait-free non-atomic double-slot buffer that tells us when it's succeeded.

We certainly haven't proceeded by "a sequence of true understatements" (Lakatos); we have made at least one "false overstatement";

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free) to an atomic double-slot buffer (ditto) to a faulty not so completely atomic double-slot buffer (still not wait-free) to a working wait-free non-atomic double-slot buffer that tells us when it's succeeded.

We certainly haven't proceeded by "a sequence of true understatements" (Lakatos); we have made at least one "false overstatement"; perhaps we have made a step of "exception barring".

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free) to an atomic double-slot buffer (ditto) to a faulty not so completely atomic double-slot buffer (still not wait-free) to a working wait-free non-atomic double-slot buffer that tells us when it's succeeded.

We certainly haven't proceeded by "a sequence of true understatements" (Lakatos); we have made at least one "false overstatement"; perhaps we have made a step of "exception barring".

I see the step that includes the '*ok*' variable as an example of thesis (program) plus antithesis (counter-example) yielding synthesis (repaired program).

Summary so far, and a criticism

We have gone from an atomic single-slot buffer (not wait-free) to an atomic double-slot buffer (ditto) to a faulty not so completely atomic double-slot buffer (still not wait-free) to a working wait-free non-atomic double-slot buffer that tells us when it's succeeded.

We certainly haven't proceeded by "a sequence of true understatements" (Lakatos); we have made at least one "false overstatement"; perhaps we have made a step of "exception barring".

I see the step that includes the '*ok*' variable as an example of thesis (program) plus antithesis (counter-example) yielding synthesis (repaired program).

But what use is the pair (false, *something*)? What can a user do but ignore *something* and try to read again?

A more honest repair

local $c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle$ in

$$
\left(
\begin{array}{l||l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \textsf{local } wt \textsf{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle; \\
\quad \textsf{ni} \\
\\ \\ \\ \\
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \textsf{local } y, rt \textsf{ in} \\
\qquad \textsf{do} \\
\qquad\quad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad\quad y := c[rt]; \\
\qquad\quad \langle\!\langle rt = ok \rangle\!\rangle \\
\qquad \textsf{until } rt; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \textsf{return } y \\
\quad \textsf{ni}
\end{array}
\end{array}
\right)
$$

A more honest repair

$$
\text{local } c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
$$

$$
\left(
\begin{array}{l||l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat=} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle; \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat=} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \text{do} \\
\qquad\quad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad\quad y := c[rt]; \\
\qquad\quad \langle\!\langle rt = ok \rangle\!\rangle \\
\qquad \text{until } rt; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}
\right)
$$

Obviously not wait-free.

A more honest repair

local $c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle$ in

$$\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle; \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \text{do} \\
\qquad\quad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad\quad y := c[rt]; \\
\qquad\quad \langle\!\langle rt = ok \rangle\!\rangle \\
\qquad \text{until } rt; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}\right)$$

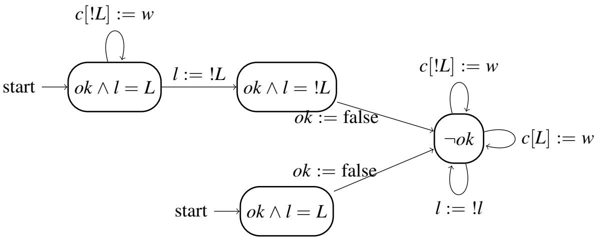
Obviously not wait-free. But otherwise repaired.

A more honest repair

$$
\text{local } c[2] = (null, null), l = 0, ok, {\color{red}ws} = \langle\,\rangle.null, {\color{red}rs} = \langle\,\rangle \text{ in}
$$

$$
\left(
\begin{array}{l||l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \text{do} \\
\qquad c[wt] := w; & \qquad\quad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt; {\color{red}ws} := {\color{red}ws}.w \rangle\!\rangle; & \qquad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle; & \qquad\quad y := c[rt]; \\
\quad \text{ni} & \qquad\quad \langle\!\langle rt = ok \rangle\!\rangle \\
 & \qquad \text{until } rt; \\
 & \qquad \langle\!\langle {\color{red}rs} := {\color{red}rs}.y \rangle\!\rangle; \\
 & \qquad \text{return } y \\
 & \quad \text{ni}
\end{array}
\right)
$$

Obviously not wait-free. But otherwise repaired. Finite-state machine now



Try three slots

$$
\text{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
$$

$$
\left(
\begin{array}{l||l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle d := w \rangle; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle; \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad y := c[rt]; \\
\qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \text{if } \neg rt \text{ then } \langle y := d \rangle \text{ else skip fi}; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}
\right)
$$

The writer can first write in a side-channel, then signal that mayhem approaches. If it gets the signal, the reader uses the side-channel.

Try three slots

$$\text{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\text{write}(w) \mathrel{\hat{=}} & \text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle ok := \text{true} \rangle\!\rangle; \\
\qquad c[wt] := w; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \qquad y := c[rt]; \\
\qquad \langle d := w \rangle; & \qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \text{false} \rangle\!\rangle; & \qquad \text{if } \neg rt \text{ then } \langle y := d \rangle \text{ else skip fi}; \\
\quad \text{ni} & \qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
 & \qquad \text{return } y \\
 & \quad \text{ni}
\end{array}\right)$$

The writer can first write in a side-channel, then signal that mayhem approaches. If it gets the signal, the reader uses the side-channel.

Perhaps this will work ...

Try three slots

$$\text{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; \\
\qquad \text{if } wt \text{ then } \langle d := w \rangle; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \\
\qquad\quad \text{else skip fi} \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad y := c[rt]; \\
\qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \text{if } \neg rt \text{ then } \langle y := d \rangle \text{ else skip fi}; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array} \right)$$

The writer can first write in a side-channel, then signal that mayhem approaches. If it gets the signal, the reader uses the side-channel.

Perhaps this will work ... but it's more likely to work if the writer only writes when the reader is asking for it.

Try three slots

$$
\textsf{local}\ c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle\ \textsf{in}
$$

$$
\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \textsf{local}\ wt\ \textsf{in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; \\
\qquad \textsf{if}\ wt\ \textsf{then}\ d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \\
\qquad\quad \textsf{else}\ \mathrm{skip}\ \textsf{fi} \\
\quad \textsf{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \textsf{local}\ y, rt\ \textsf{in} \\
\qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad y := c[rt]; \\
\qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \textsf{if}\ \neg rt\ \textsf{then}\ y := d\ \textsf{else}\ \mathrm{skip}\ \textsf{fi}; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \textsf{return}\ y \\
\quad \textsf{ni}
\end{array}
\end{array}\right)
$$

The writer can first write in a side-channel, then signal that mayhem approaches. If it gets the signal, the reader uses the side-channel.

Perhaps this will work ... but it's more likely to work if the writer only writes when the reader is asking for it.

And then I notice that they alternate, and I can use *non*-atomic read and write.

Try three slots

$$
\text{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
$$

$$
\left(
\begin{array}{l||l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat=} & \mathrm{read}(\,) \mathrel{\hat=} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := \,!l \rangle\!\rangle; & \qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad c[wt] := w; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \qquad y := c[rt]; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; & \qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \text{if } wt \text{ then } d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle & \qquad \text{if } \neg rt \text{ then } y := d \text{ else } \mathrm{skip} \text{ fi}; \\
\qquad\quad \text{else } \mathrm{skip} \text{ fi} & \qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\quad \text{ni} & \qquad \text{return } y \\
 & \quad \text{ni}
\end{array}
\right)
$$

The writer can first write in a side-channel, then signal that mayhem approaches. If it gets the signal, the reader uses the side-channel.

Perhaps this will work ... but it's more likely to work if the writer only writes when the reader is asking for it.

And then I notice that they alternate, and I can use *non*-atomic read and write. This is Harris's algorithm, rationally developed.

Recapitulation

$$
\begin{array}{l}
\textsf{local } b = \mathit{null} \textsf{ in} \\
\left(\begin{array}{l|l}
\quad \ldots & \quad \ldots \\
\quad \mathrm{write}(w) \mathrel{\hat{=}} & \quad \mathrm{read}(\,) \mathrel{\hat{=}} \\
\qquad \langle b := w \rangle & \qquad \textsf{local } y \textsf{ in} \\
& \qquad\quad \langle y := b \rangle; \\
& \qquad\quad \textsf{return } y \\
& \qquad \textsf{ni}
\end{array}\right)
\end{array}
$$

– Dijkstra's single-place buffer.

Recapitulation

$$
\begin{array}{l}
\textsf{local } b = null, ws = \langle\,\rangle.null, rs = \langle\,\rangle \textsf{ in} \\
\left(\begin{array}{l|l}
\quad \ldots & \ldots \\
\quad \mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\
\qquad \langle b := w; ws := ws.w \rangle & \quad \textsf{local } y \textsf{ in} \\
 & \qquad \langle y := b; rs := rs.y \rangle; \\
 & \qquad \textsf{return } y \\
 & \quad \textsf{ni}
\end{array}\right)
\end{array}
$$

– with auxiliary ws and rs to show $\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b$.

Recapitulation

$$\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l} \ldots \\ \mathrm{write}(w) \mathrel{\hat{=}} \\ \quad \left\langle \begin{array}{l} c[!l] := w; l := !l; \\ ws := ws.w \end{array} \right\rangle \\ \\ \end{array} \;\middle\|\; \begin{array}{l} \ldots \\ \mathrm{read}(\,) \mathrel{\hat{=}} \\ \quad \text{local } y \text{ in} \\ \quad\quad \langle y := c[l]; rs := rs.y \rangle; \\ \quad\quad \text{return } y \\ \quad \text{ni} \end{array} \right)$$

– data refinement to two slots and $\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l]$.

Recapitulation

$$
\begin{array}{l}
\textsf{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \textsf{ in} \\
\left(
\begin{array}{l||l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \textsf{local } wt \textsf{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \\
\quad \textsf{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \textsf{local } y \textsf{ in} \\
\qquad \langle y := c[l]; rs := rs.y \rangle; \\
\qquad \textsf{return } y \\
\quad \textsf{ni} \\
\end{array}
\end{array}
\right)
\end{array}
$$

– atomicity refinement in the writer; still $\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l]$.

Recapitulation

$$
\text{local } c[2] = (null, null), l = 0, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
$$

$$
\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\text{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad \langle c[wt] := w \rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\text{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle y := c[rt]; rs := rs.y \rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}\right)
$$

– atomicity refinement in the reader; now $\lfloor rs \rfloor \not\preccurlyeq ws \wedge ws_\Omega = c[l]$.

Recapitulation

$$
\text{local } c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
$$

$$
\left(\begin{array}{l|l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle; \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad y := c[rt]; \\
\qquad \langle\!\langle rt = ok; rs := rs.(rt, y) \rangle\!\rangle; \\
\qquad \text{return } (rt, y) \\
\quad \text{ni}
\end{array}
\end{array}\right)
$$

– exception barring; now $\lfloor \widetilde{rs} \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l]$.

Recapitulation

$$
\text{local } c[2] = (null, null), l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}
$$

$$
\left(\begin{array}{l||l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle; \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \text{do} \\
\qquad\quad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad\quad y := c[rt]; \\
\qquad\quad \langle\!\langle rt = ok \rangle\!\rangle \\
\qquad \text{until } rt; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}\right)
$$

– more honest exception barring; once again $\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l]$, but no longer wait-free.

Recapitulation

$$\text{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad c[wt] := w; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \qquad y := c[rt]; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; & \qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \text{if } wt \text{ then } d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle & \qquad \text{if } \neg rt \text{ then } y := d \text{ else } \mathrm{skip} \text{ fi}; \\
\qquad\quad \text{else } \mathrm{skip} \text{ fi} & \qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\quad \text{ni} & \qquad \text{return } y \\
 & \quad \text{ni}
\end{array}\right)$$

– three slots; $\lfloor rs \rfloor \preccurlyeq ws \wedge ws_{\Omega} = c[l]$; wait-free.

Recapitulation

$$\mathsf{local}\ c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle\ \mathsf{in}$$

$$\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \mathsf{local}\ wt\ \mathsf{in} & \quad \mathsf{local}\ y, rt\ \mathsf{in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad c[wt] := w; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \qquad y := c[rt]; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; & \qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \mathsf{if}\ wt\ \mathsf{then}\ d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle & \qquad \mathsf{if}\ \neg rt\ \mathsf{then}\ y := d\ \mathsf{else}\ \mathrm{skip}\ \mathsf{fi}; \\
\qquad\quad \mathsf{else}\ \mathrm{skip}\ \mathsf{fi} & \qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\quad \mathsf{ni} & \qquad \mathsf{return}\ y \\
 & \quad \mathsf{ni}
\end{array}\right)$$

– three slots; $\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l]$; wait-free.

Proof available on application.

A programmer's instinct demands ...

$$\text{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat=} & \mathrm{read}(\,) \mathrel{\hat=} \\
\quad \text{local } wt \text{ in} & \quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad c[wt] := w; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \qquad y := c[rt]; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; & \qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \text{if } wt \text{ then } d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle & \qquad \text{if } \neg rt \text{ then } y := d \\
\qquad\quad \text{else } \mathrm{skip} \text{ fi} & \qquad\quad \text{else } \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \text{ fi}; \\
\quad \text{ni} & \qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
 & \qquad \text{return } y \\
 & \quad \text{ni}
\end{array}\right)$$

– the reader tells the writer when there's no need to use the side channel.

A programmer's instinct demands ...

local $c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle$ in

$$
\left(
\begin{array}{l||l}
\ldots & \ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} & \mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \textsf{local } wt \textsf{ in} & \quad \textsf{local } y, rt \textsf{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad c[wt] := w; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \qquad y := c[rt]; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; & \qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \textsf{if } wt \textsf{ then } d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle & \qquad \textsf{if } \neg rt \textsf{ then } y := d \\
\qquad\quad \textsf{else } \mathrm{skip} \textsf{ fi} & \qquad\quad \textsf{else } \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \textsf{ fi}; \\
\quad \textsf{ni} & \qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
 & \qquad \textsf{return } y \\
 & \quad \textsf{ni}
\end{array}
\right)
$$

– the reader tells the writer when there's no need to use the side channel. It's no longer true that the writer only writes when *ok*.

A programmer's instinct demands . . .

$$\text{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \text{ in}$$

$$\left(\begin{array}{l||l}
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \text{local } wt \text{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; \\
\qquad \text{if } wt \text{ then } d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \\
\qquad\quad \text{else } \mathrm{skip} \text{ fi} \\
\quad \text{ni}
\end{array}
&
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \text{local } y, rt \text{ in} \\
\qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad y := c[rt]; \\
\qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \text{if } \neg rt \text{ then } y := d \\
\qquad\quad \text{else } \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \text{ fi}; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \text{return } y \\
\quad \text{ni}
\end{array}
\end{array}\right)$$

– the reader tells the writer when there's no need to use the side channel. It's no longer true that the writer only writes when *ok*. But from the point of view of the reader, nothing has changed!

A programmer's instinct demands …

$$
\begin{array}{l}
\textsf{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \textsf{ in}\\
\left(\begin{array}{l|l}
\begin{array}{l}
\ldots\\
\mathrm{write}(w) \mathrel{\hat{=}}\\
\quad \textsf{local } wt \textsf{ in}\\
\qquad \langle\!\langle wt := !l \rangle\!\rangle;\\
\qquad c[wt] := w;\\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle;\\
\qquad \langle\!\langle wt := ok \rangle\!\rangle;\\
\qquad \textsf{if } wt \textsf{ then } d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle\\
\qquad\quad \textsf{else } \mathrm{skip} \textsf{ fi}\\
\quad \textsf{ni}
\end{array}
&
\begin{array}{l}
\ldots\\
\mathrm{read}(\,) \mathrel{\hat{=}}\\
\quad \textsf{local } y, rt \textsf{ in}\\
\qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle;\\
\qquad \langle\!\langle rt := l \rangle\!\rangle;\\
\qquad y := c[rt];\\
\qquad \langle\!\langle rt = ok \rangle\!\rangle;\\
\qquad \textsf{if } \neg rt \textsf{ then } y := d\\
\qquad\quad \textsf{else } \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \textsf{ fi};\\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle;\\
\qquad \textsf{return } y\\
\quad \textsf{ni}
\end{array}
\end{array}\right)
\end{array}
$$

– the reader tells the writer when there's no need to use the side channel. It's no longer true that the writer only writes when *ok*. But from the point of view of the reader, nothing has changed!

In fact the writer writes when *ok* <u>or</u> the reader is asleep.

A programmer's instinct demands . . .

$$
\begin{array}{l}
\textsf{local } c[2] = (null, null), d = null, l = 0, ok, ws = \langle\,\rangle.null, rs = \langle\,\rangle \textsf{ in} \\
\left(
\begin{array}{l}
\ldots \\
\mathrm{write}(w) \mathrel{\hat{=}} \\
\quad \textsf{local } wt \textsf{ in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; \\
\qquad c[wt] := w; \\
\qquad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; \\
\qquad \langle\!\langle wt := ok \rangle\!\rangle; \\
\qquad \textsf{if } wt \textsf{ then } d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \\
\qquad\quad \textsf{else } \mathrm{skip} \textsf{ fi} \\
\quad \textsf{ni}
\end{array}
\;\middle\|\;
\begin{array}{l}
\ldots \\
\mathrm{read}(\,) \mathrel{\hat{=}} \\
\quad \textsf{local } y, rt \textsf{ in} \\
\qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad y := c[rt]; \\
\qquad \langle\!\langle rt = ok \rangle\!\rangle; \\
\qquad \textsf{if } \neg rt \textsf{ then } y := d \\
\qquad\quad \textsf{else } \langle\!\langle ok := \mathrm{false} \rangle\!\rangle \textsf{ fi}; \\
\qquad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
\qquad \textsf{return } y \\
\quad \textsf{ni}
\end{array}
\right)
\end{array}
$$

– the reader tells the writer when there's no need to use the side channel. It's no longer true that the writer only writes when *ok*. But from the point of view of the reader, nothing has changed!

In fact the writer writes when *ok* <u>or</u> the reader is asleep. Easy to fix with another auxiliary variable, proof available on request.

Pipelines, caches, weak memory models

In my argument I relied on sequencing of actions and on serialisation of (small) memory accesses.

Pipelines, caches, weak memory models

In my argument I relied on sequencing of actions and on serialisation of (small) memory accesses.

In modern machines these assumptions don't hold.

Pipelines, caches, weak memory models

In my argument I relied on sequencing of actions and on serialisation of (small) memory accesses.

In modern machines these assumptions don't hold.

Pipelines reorder instructions if there isn't an obvious dependency – e.g. in $c[wt] := w; l := wt$ or in $d := w; ok :=$ false.

Pipelines, caches, weak memory models

In my argument I relied on sequencing of actions and on serialisation of (small) memory accesses.

In modern machines these assumptions don't hold.

Pipelines reorder instructions if there isn't an obvious dependency – e.g. in $c[wt] := w; l := wt$ or in $d := w; ok :=$ false.

Caches delay interaction with the store.

Pipelines, caches, weak memory models

In my argument I relied on sequencing of actions and on serialisation of (small) memory accesses.

In modern machines these assumptions don't hold.

Pipelines reorder instructions if there isn't an obvious dependency – e.g. in $c[wt] := w; l := wt$ or in $d := w; ok :=$ false.

Caches delay interaction with the store. Viewed from another machine, store actions are reordered.

Pipelines, caches, weak memory models

In my argument I relied on sequencing of actions and on serialisation of (small) memory accesses.

In modern machines these assumptions don't hold.

Pipelines reorder instructions if there isn't an obvious dependency – e.g. in $c[wt] := w; l := wt$ or in $d := w; ok :=$ false.

Caches delay interaction with the store. Viewed from another machine, store actions are reordered.

So-called "weak memory models" are difficult to understand.

Pipelines, caches, weak memory models

In my argument I relied on sequencing of actions and on serialisation of (small) memory accesses.

In modern machines these assumptions don't hold.

Pipelines reorder instructions if there isn't an obvious dependency – e.g. in $c[wt] := w; l := wt$ or in $d := w; ok :=$ false.

Caches delay interaction with the store. Viewed from another machine, store actions are reordered.

So-called "weak memory models" are difficult to understand. That is, to understand well enough to reason about programs running under them.

Pipelines, caches, weak memory models

In my argument I relied on sequencing of actions and on serialisation of (small) memory accesses.

In modern machines these assumptions don't hold.

Pipelines reorder instructions if there isn't an obvious dependency – e.g. in $c[wt] := w; l := wt$ or in $d := w; ok :=$ false.

Caches delay interaction with the store. Viewed from another machine, store actions are reordered.

So-called "weak memory models" are difficult to understand. That is, to understand well enough to reason about programs running under them.

Maybe an EPSRC project . . .

And finally

Programming is creative, experimental.

And finally

Programming is creative, experimental.

That means we will go wrong.

And finally

Programming is creative, experimental.

That means we will go wrong.

When we go wrong we can fix our program, refine our conjecture, or both.

And finally

Programming is creative, experimental.

That means we will go wrong.

When we go wrong we can fix our program, refine our conjecture, or both.

The mechanisms of refinement can guide us, but we may (I would say must) sometimes make “false overstatements”.

And finally

Programming is creative, experimental.

That means we will go wrong.

When we go wrong we can fix our program, refine our conjecture, or both.

The mechanisms of refinement can guide us, but we may (I would say must) sometimes make "false overstatements".

Lakatos's dialectic lives!