Multicore Programming

Peter Sewell Jaroslav Ševčík Tim Harris

University of Cambridge MSR

with thanks to

Francesco Zappa Nardelli, Susmit Sarkar, Tom Ridge, Scott Owens, Magnus O. Myreen, Luc Maranget, Mark Batty, Jade Alglave

October – November, 2010

Small language – expressions

$location,\ x$	address (or pointer value)	
$variable,\ r$	thread-local variable name	
$integer,\ n$	integer	
$thread\_id,\ t$	thread id	
$expression,\ e$	$::=$	term
	$\mid\ n$	integer literal
	$\mid\ *x$	read from pointer
	$\mid\ *x = e$	write to pointer
	$\mid\ r$	read from local variable
	$\mid\ r = e$	write to local variable
	$\mid\ e; e'$	sequential composition
	$\mid\ \mathbf{if}\ e = e'\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$	conditional
	$\mid\ \mathbf{lock}\ x$	lock
	$\mid\ \mathbf{unlock}\ x$	unlock
	$\mid\ \mathbf{print}\ e$	print
	$\mid\ (e)$	S

Small language – processes and states

Let memory M be a map from locations to integers and L be a set of locations.

$process,\ p$	$::=$		process
	$\mid$	$t{:}\rho;e$	expression
	$\mid$	$p \mid p'$	parallel composition
$state,\ s$	$::=$		state
	$\mid$	$\langle p,\ M,\ L \rangle$	
$label,\ l$	$::=$		label
	$\mid$	W $x{=}n$	write
	$\mid$	R $x{=}n$	read
	$\mid$	L x	lock
	$\mid$	U x	unlock
	$\mid$	P n	print
	$\mid$	τ	internal action (tau)

Expression semantics

Let ρ be a map from thread-local variable names to integers.

$\boxed{\rho_1; e_1 \xrightarrow{l} \rho_2; e_2}$ e_1 does l to become e_2

$$\frac{}{\rho; *x \xrightarrow{\mathsf{R}\, x=n} \rho; n}$$ READ

$$\frac{}{\rho; *x = n \xrightarrow{\mathsf{W}\, x=n} \rho; n}$$ WRITE

$$\frac{\rho(r) = n}{\rho; r \xrightarrow{\tau} \rho; n}$$ VAR_READ

$$\frac{}{\rho; r = n \xrightarrow{\tau} \rho \oplus (r \mapsto n); n}$$ VAR_WRITE

$$\frac{\rho; e_1 \xrightarrow{l} \rho'; e_1'}{\rho; e_1;\, e_2 \xrightarrow{l} \rho'; e_1';\, e_2}$$ SEQ_CONTEXT

Expression semantics

$$\frac{}{\rho;n;\, e \xrightarrow{\tau} \rho;e} \quad \text{SEQ}$$

$$\frac{\rho;e_1 \xrightarrow{l} \rho';e_1'}{\rho;\mathbf{if}\ e_1 = e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \xrightarrow{l} \rho';\mathbf{if}\ e_1' = e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4} \quad \text{IF\_CONTEXT\_1}$$

$$\frac{\rho;e_2 \xrightarrow{l} \rho';e_2'}{\rho;\mathbf{if}\ n = e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \xrightarrow{l} \rho';\mathbf{if}\ n = e_2'\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4} \quad \text{IF\_CONTEXT\_2}$$

$$\frac{}{\rho;\mathbf{if}\ n = n\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \xrightarrow{\tau} \rho;e_3} \quad \text{IF\_EQ}$$

$$\frac{n \neq n'}{\rho;\mathbf{if}\ n = n'\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \xrightarrow{\tau} \rho;e_4} \quad \text{IF\_NEQ}$$

$$\frac{}{\rho;\mathbf{lock}\ x \xrightarrow{\mathsf{L}\,x} \rho;0} \quad \text{LOCK}$$

Expression semantics

$$\frac{}{\rho;\mathbf{unlock}\,x \xrightarrow{\mathsf{U}\,x} \rho;0}\ \textsc{unlock}$$

$$\frac{\rho;e \xrightarrow{l} \rho';e'}{\rho;\mathbf{print}\,e \xrightarrow{l} \rho';\mathbf{print}\,e'}\ \textsc{print\_context}$$

$$\frac{}{\rho;\mathbf{print}\,n \xrightarrow{\mathsf{P}\,n} \rho;n}\ \textsc{print}$$

Process semantics

$\boxed{p_1 \xrightarrow{t:l} p_2}$ p_1 does l to become p_2

$$\frac{\rho;e \xrightarrow{l} \rho';e'}{t{:}\rho;e \xrightarrow{t:l} t{:}\rho';e'} \quad \text{THREAD}$$

$$\frac{p_1 \xrightarrow{t:l} p_1'}{p_1|p_2 \xrightarrow{t:l} p_1'|p_2} \quad \text{PAR\_CONTEXT\_LEFT}$$

$$\frac{p_2 \xrightarrow{t:l} p_2'}{p_1|p_2 \xrightarrow{t:l} p_1|p_2'} \quad \text{PAR\_CONTEXT\_RIGHT}$$

SC semantics

$\boxed{s_1 \xrightarrow{t:l} s_2}$ s_1 makes a step to become s_2

$$\frac{p \xrightarrow{t:\mathsf{R}\,x=n} p' \qquad M(x) = n}{\langle p,\ M,\ L\rangle \xrightarrow{t:\mathsf{R}\,x=n} \langle p',\ M,\ L\rangle}\ \textsc{Sread}$$

$$\frac{p \xrightarrow{t:\mathsf{W}\,x=n} p'}{\langle p,\ M,\ L\rangle \xrightarrow{t:\mathsf{W}\,x=n} \langle p',\ M \oplus (x \mapsto n),\ L\rangle}\ \textsc{Swrite}$$

$$\frac{p \xrightarrow{t:\tau} p'}{\langle p,\ M,\ L\rangle \xrightarrow{t:\tau} \langle p',\ M,\ L\rangle}\ \textsc{Stau}$$

$$\frac{p \xrightarrow{t:\mathsf{L}\,x} p' \qquad x \notin L}{\langle p,\ M,\ L\rangle \xrightarrow{t:\mathsf{L}\,x} \langle p',\ M,\ L \cup \{x\}\rangle}\ \textsc{Slock}$$

$$\frac{p \xrightarrow{t:\mathsf{U}\,x} p'}{\langle p,\ M,\ L\rangle \xrightarrow{t:\mathsf{U}\,x} \langle p',\ M,\ L \setminus \{x\}\rangle}\ \textsc{Sunlock}$$

$$\frac{p \xrightarrow{t:\mathsf{P}\,n} p'}{\langle p,\ M,\ L\rangle \xrightarrow{t:\mathsf{P}\,n} \langle p',\ M,\ L\rangle}\ \textsc{Sprint}$$

- Compiler Optimisations
  - Correctness
- Optimisations in SC
- Optimisations of Data-race-free Programs
  - Data race freedom
  - Limitations of the DRF principle
  - Correctness under the DRF principle
- Out-of-thin-air values

Compiler Optimisations

- Correctness
- Optimisations in SC
- Optimisations of Data-race-free Programs
  - Data race freedom
  - Limitations of the DRF principle
  - Correctness under the DRF principle
- Out-of-thin-air values

World of Optimisations

Typical compiler performs many optimisations.

For example, `gcc` 4.4.1. with `-O2` option goes through 147 compilation passes (using `-fdump-tree-all` and `-fdump-rtl-all`).

Sun Hotspot Server JVM has 18 high-level passes with each pass composed of one or more smaller passes (`http://www.azulsystems.com/blog/cliff-click/2009-04-14-odds-ends`).

World of Optimisations

Typical compiler performs many optimisations.

– Common subexpression elimination (copy propagation, partial redundancy elimination, value numbering)
– (conditional) constant propagation
– dead code elimination
– loop optimisations (loop invariant code motion, loop splitting/peeling, loop unrolling, etc.)
– vectorisation
– peephole optimisations
– tail duplication removal
– building graph representations/graph linearisation
– register allocation
– call inlining
– local memory to registers promotion
– spilling
– instruction scheduling. . .

Memory Optimisations

Only some optimisations change shared-memory traces.

Memory Optimisations

Only some optimisations change shared-memory traces.

– Common subexpression elimination (copy propagation, partial redundancy elimination, value numbering)
– (conditional) constant propagation
– dead code elimination
– loop optimisations (loop invariant code motion, loop splitting/peeling, loop unrolling, etc.)
– vectorisation
– peephole optimisations
– tail duplication removal
– building graph representations/graph linearisation
– register allocation
– call inlining
– local memory to registers promotion
– spilling
– instruction scheduling. . .

Memory Optimisations

Only some optimisations change shared-memory traces.

The optimisations of shared-memory can perform:

- Eliminations (of reads, writes, sometimes synchronisation).
- Reordering (of independent non-conflicting memory accesses).
- Introductions (of reads – rarely).

Eliminations

This includes common subexpression elimination, dead read elimination, overwritten write elimination, redundant write elimination.

- Irrelevant read elimination:

$$\texttt{r=*x;}\ C \rightarrow C,$$

  where `r` is not free in C.

- Redundant read after read elimination:

$$\texttt{r1=*x; r2=*x} \rightarrow \texttt{r1=*x; r2=r1}.$$

- Redundant read after write elimination:

$$\texttt{*x=r1; r2=*x} \rightarrow \texttt{*x=r1; r2=r1}.$$

Reordering

Some loop optimisations, code motion.

- Normal memory access reordering:

$$\texttt{r1=*x; r2=*y} \longrightarrow \texttt{r2=*y; r1=*x},$$
$$\texttt{*x=r1; *y=r2} \longrightarrow \texttt{*y=r2; *x=r1},$$
$$\texttt{r1=*x; *y=r2} \rightleftarrows \texttt{*y=r2; r1=*x}.$$

- Roach motel reordering:

$$\texttt{memop; lock m} \longrightarrow \texttt{lock m; memop},$$
$$\texttt{unlock m; memop} \longrightarrow \texttt{memop; unlock m},$$

where `memop` is `*x=r1` or `r1=*x`.

Memory access introductions

Can an optimisation introduce memory accesses?

Note that the loop body is not executed.

Memory access introductions

Can an optimisation introduce memory accesses?

Yes, but rarely:

```
i = 0;
...
while (i != 0) (
  j = *x + 1;
  i = i - 1 )
```

→

```
i = 0;
...
tmp = *x;
while (i != 0) (
  j = tmp + 1;
  i = i - 1 )
```

Note that the loop body is not executed.

Compiler Optimisations

Correctness

Optimisations in SC

Optimisations of Data-race-free Programs

- Data race freedom
- Limitations of the DRF principle
- Correctness under the DRF principle

Out-of-thin-air values

When is an optimisation correct?

Optimisation is correct if any behaviour of the optimised program could be exhibited by the original program.

I.e., for any execution of the optimised program, there is an execution of the original program with the same observable behaviour.

When is an optimisation correct?

Optimisation is correct if any behaviour of the optimised program could be exhibited by the original program.

I.e., for any execution of the optimised program, there is an execution of the original program with the same observable behaviour.

Where the observable behaviour of an execution is the subtrace of external actions (let us ignore termination).

Example

$P_1 =$ `*x = 1` | `r1 = *x; r2 = *x;`
`if r1=r2 then print 1 else print 2`

$P_2 =$ `*x = 1` | `r1 = *x; r2 = r1;`
`if r1=r2 then print 1 else print 2`

Is the transformation from P_1 to P_2 correct (in our SC semantics)?

Example

$P_1 =$ `*x = 1` | `r1 = *x; r2 = *x;`
`if r1=r2 then print 1 else print 2`

$P_2 =$ `*x = 1` | `r1 = *x; r2 = r1;`
`if r1=r2 then print 1 else print 2`

Executions of P_1:

$\mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 2$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{P}_{t_2}\, 1$
$\mathsf{R}_{t_2}\, x{=}0, \mathsf{R}_{t_2}\, x{=}0, \mathsf{P}_{t_2}\, 1, \mathsf{W}_{t_1}\, x{=}1$

Example

$P_1 =$ `*x = 1` | `r1 = *x; r2 = *x;` `if r1=r2 then print 1 else print 2`

$P_2 =$ `*x = 1` | `r1 = *x; r2 = r1;` `if r1=r2 then print 1 else print 2`

Executions of P_2:

$\mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{P}_{t_2}\, 1$

$\mathsf{R}_{t_2}\, x{=}0, \mathsf{W}_{t_1}\, x{=}1, \mathsf{P}_{t_2}\, 1$

$\mathsf{R}_{t_2}\, x{=}0, \mathsf{P}_{t_2}\, 1, \mathsf{W}_{t_1}\, x{=}1$

Example

$P_1 =$ `*x = 1` | `r1 = *x; r2 = *x;`
`if r1=r2 then print 1 else print 2`

$P_2 =$ `*x = 1` | `r1 = *x; r2 = r1;`
`if r1=r2 then print 1 else print 2`

Behaviours of P_1: $[\mathsf{P}_{t_2}\, 1], [\mathsf{P}_{t_2}\, 2]$.

Behaviours of P_2: $[\mathsf{P}_{t_2}\, 1]$.

It is correct to rewrite $P_1 \rightarrow P_2$, but not the opposite!

Optimisation Correctness Overview

Transformation	SC	JMM	DRF
Trace-preserving transformations	✓	✓	✓
Reordering normal memory accesses	×	×	✓
Redundant read after read elimination	✓*	×	✓
Redundant read after write elimination	✓*	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	×	?
Redundant write before write elimination	✓*	✓	✓
Redundant write after read elimination	✓*	×	✓
Roach-motel reordering	✓	×	✓
External action reordering	×	×	✓

✓– correct, ×– incorrect, ✓* – correct only for adjacent memory accesses.

Correctness proof strategy

Take an arbitrary execution of the optimised program.

Correctness proof strategy

Take an arbitrary execution of the optimised program.

Massage it into an execution of the original program. . .

Correctness proof strategy

Take an arbitrary execution of the optimised program.

Massage it into an execution of the original program. . .

. . . so that the massaged execution has the same sequence of observable actions.

Memory actions are **not** observable.

Compiler Optimisations

- Correctness

Optimisations in SC

Optimisations of Data-race-free Programs

- Data race freedom
- Limitations of the DRF principle
- Correctness under the DRF principle

Out-of-thin-air values

General CSE incorrect in SC

```
*x = 1;          |  if *y=1 then (
*y = 1;          |    *x = 2;
if *y = 2        |    *y = 2
then print *x    |  )
```

There is only one execution with a printing behaviour:

$\mathsf{W}_{t_1}\, x{=}1, \mathsf{W}_{t_1}\, y{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{W}_{t_2}\, x{=}2, \mathsf{W}_{t_2}\, y{=}2, \mathsf{R}_{t_1}\, y{=}2, \mathsf{R}_{t_1}\, x{=}2, \mathsf{P}_{t_1}\, 2$

General CSE incorrect in SC

```
*x = 1;          |  if *x=1 then (
*y = 1;          |    *x = 2;
if *y = 2        |    *y = 2
then print *x    |  )
```

But a compiler would optimise to:

```
*x = 1;          |  if *x=1 then (
*y = 1;          |    *x = 2;
if *y = 2        |    *y = 2
then print 1     |  )
```

General CSE incorrect in SC

```
*x = 1;          |  if *x=1 then (
*y = 1;          |    *x = 2;
if *y = 2        |    *y = 2
then print 1     |  )
```

The only execution with a printing behaviour in the optimised program is:

$$\mathsf{W}_{t_1}\, x{=}1, \mathsf{W}_{t_1}\, y{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{W}_{t_2}\, x{=}2, \mathsf{W}_{t_2}\, y{=}2, \mathsf{R}_{t_1}\, y{=}2, \mathsf{P}_{t_1}\, 1$$

So the optimisation is not correct!

General CSE incorrect in SC II

`*x = 1;`	`r = *x;`
`*y = 1;`	`print r;`
	`print *y;`
	`print *x;`

The observable behaviours are:

$$[\mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,1]$$
$$[\mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,1]$$
$$[\mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,1]$$
$$[\mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,1]$$
$$[\mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,0]$$

Note that $[\mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,0]$ is not observable.

General CSE incorrect in SC II

```
*x = 1;  |  r = *x;
*y = 1;  |  print r;
         |  print *y;
         |  print *x;
```

But a compiler would optimise to:

```
*x = 1;  |  r = *x;
*y = 1;  |  print r;
         |  print *y;
         |  print r;
```

General CSE incorrect in SC II

The optimised program

```
*x = 1;  |  r = *x;
*y = 1;  |  print r;
         |  print *y;
         |  print r;
```

has behaviours:

$[\mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,1]$

$[\mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,1]$

$[\mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,1, \mathsf{P}_{t_2}\,0]$ New behaviour!

$[\mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,0, \mathsf{P}_{t_2}\,0]$

Reordering incorrect

`*x = 1;`	`*y = 1;`		`r1 = *y`	`*y = 1;`
`r1 = *y`	`r2 = *x;`	$\Rightarrow$	`*x = 1;`	`r2 = *x;`
`print r1`	`print r2`		`print r1`	`print r2`

Behaviours:

$[\mathsf{P}_{t_1}\ 0, \mathsf{P}_{t_2}\ 1]$
$[\mathsf{P}_{t_1}\ 1, \mathsf{P}_{t_2}\ 0]$
$[\mathsf{P}_{t_1}\ 1, \mathsf{P}_{t_2}\ 1]$

Behaviours:

$[\mathsf{P}_{t_1}\ 0, \mathsf{P}_{t_2}\ 1]$
$[\mathsf{P}_{t_1}\ 1, \mathsf{P}_{t_2}\ 0]$
$[\mathsf{P}_{t_1}\ 1, \mathsf{P}_{t_2}\ 1]$
$[\mathsf{P}_{t_1}\ 0, \mathsf{P}_{t_2}\ 0]$

This is essentially the same example as in the first x86 lecture.

Elimination of Adjacent Accesses

There are some correct optimisations under SC. For example, it is correct to rewrite

```
r1 = *x; r2 = *x    →    r1 = *x; r2 = r1
```

in any context.

Why?

Elimination of Adjacent Accesses

There are some correct optimisations under SC. For example, it is correct to rewrite

```
r1 = *x; r2 = *x    →    r1 = *x; r2 = r1
```

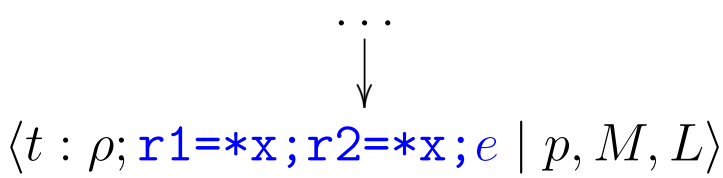
in any context.

Why?

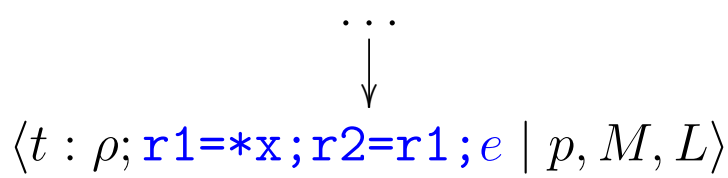
The basic idea is: whenever we perform the read `r1 = *x` in the transformed program, we perform both reads in the original program.

Simulating Elimination

Original program



Optimised program



Simulating Elimination

Original program

$$\vdots$$
$$\downarrow$$
$$\langle t : \rho; \texttt{r1=*x;r2=*x;}e \mid p, M, L\rangle$$

Optimised program

$$\vdots$$
$$\downarrow$$
$$\langle t : \rho; \texttt{r1=*x;r2=r1;}e \mid p, M, L\rangle$$
$$\downarrow \mathsf{R}_t\ x{=}n$$
$$\langle t : \rho(r1 \mapsto n); \texttt{r2=r1;}e \mid p, M, L\rangle$$

Simulating Elimination

Original program

$$\vdots$$
$$\downarrow$$
$$\langle t : \rho; \texttt{r1=*x;r2=*x;}e \mid p, M, L\rangle$$
$$\downarrow \mathsf{R}_t\, x{=}n$$
$$\langle t : \rho(r1 \mapsto n); \texttt{r2=*x;}e \mid p, M, L\rangle$$
$$\downarrow \mathsf{R}_t\, x{=}n$$
$$\langle t : \rho\begin{pmatrix} r1 \mapsto n \\ r2 \mapsto n \end{pmatrix}; e \mid p, M, L\rangle$$

Optimised program

$$\vdots$$
$$\downarrow$$
$$\langle t : \rho; \texttt{r1=*x;r2=r1;}e \mid p, M, L\rangle$$
$$\downarrow \mathsf{R}_t\, x{=}n$$
$$\langle t : \rho(r1 \mapsto n); \texttt{r2=r1;}e \mid p, M, L\rangle$$

Simulating Elimination

Original program

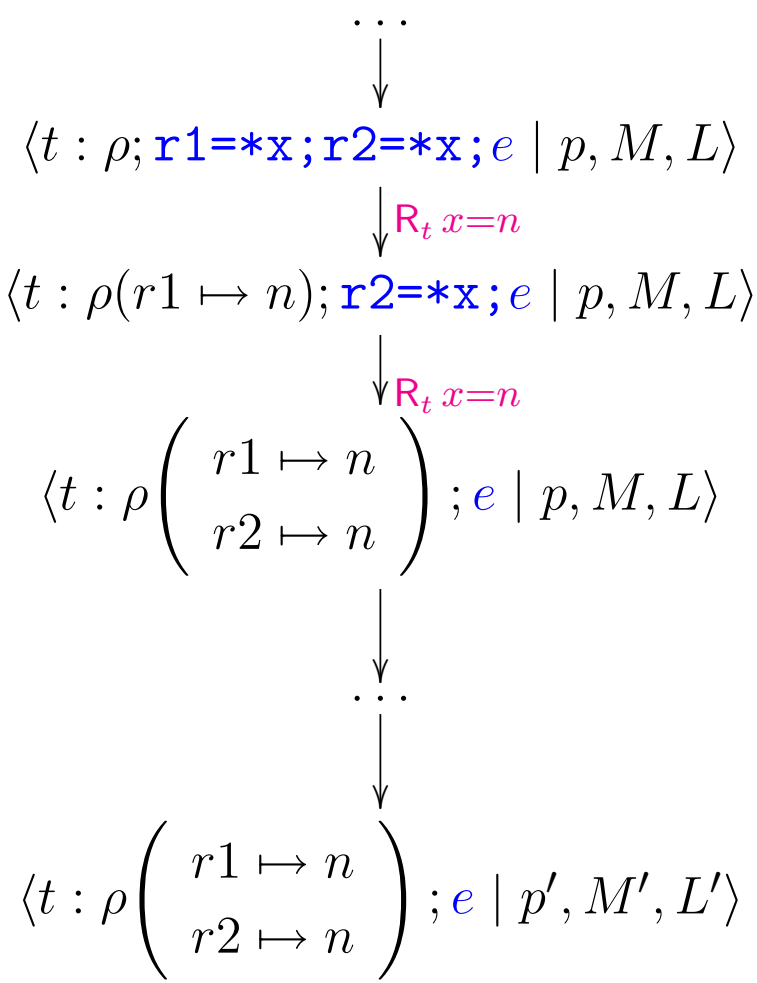
$$\vdots$$
$$\downarrow$$
$$\langle t : \rho; \texttt{r1=*x;r2=*x;}e \mid p, M, L\rangle$$
$$\downarrow \mathsf{R}_t\ x{=}n$$
$$\langle t : \rho(r1 \mapsto n); \texttt{r2=*x;}e \mid p, M, L\rangle$$
$$\downarrow \mathsf{R}_t\ x{=}n$$
$$\langle t : \rho\begin{pmatrix} r1 \mapsto n \\ r2 \mapsto n \end{pmatrix}; e \mid p, M, L\rangle$$

Optimised program

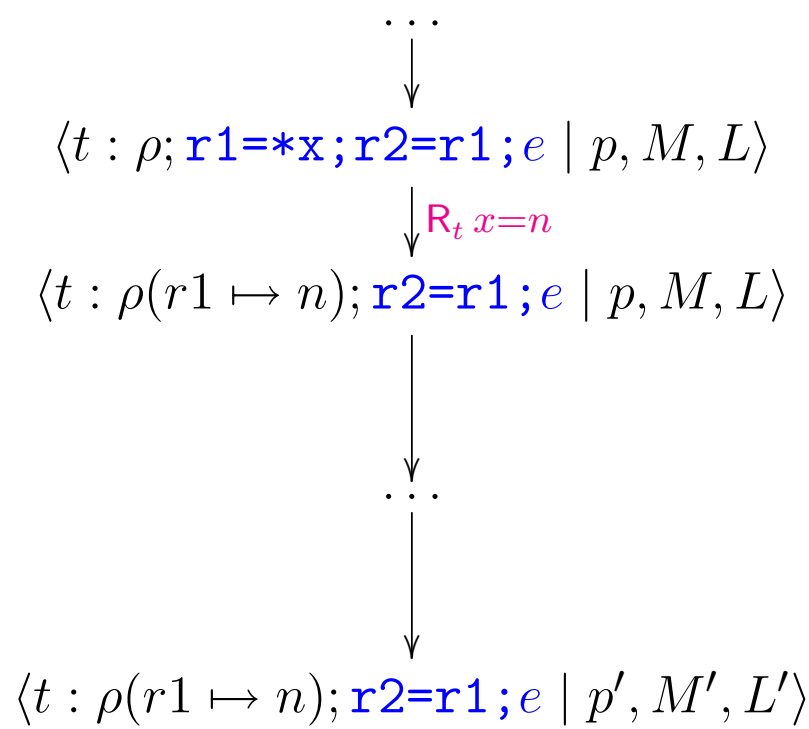
$$\vdots$$
$$\downarrow$$
$$\langle t : \rho; \texttt{r1=*x;r2=r1;}e \mid p, M, L\rangle$$
$$\downarrow \mathsf{R}_t\ x{=}n$$
$$\langle t : \rho(r1 \mapsto n); \texttt{r2=r1;}e \mid p, M, L\rangle$$
$$\downarrow$$
$$\vdots$$
$$\downarrow$$
$$\langle t : \rho(r1 \mapsto n); \texttt{r2=r1;}e \mid p', M', L'\rangle$$

Simulating Elimination

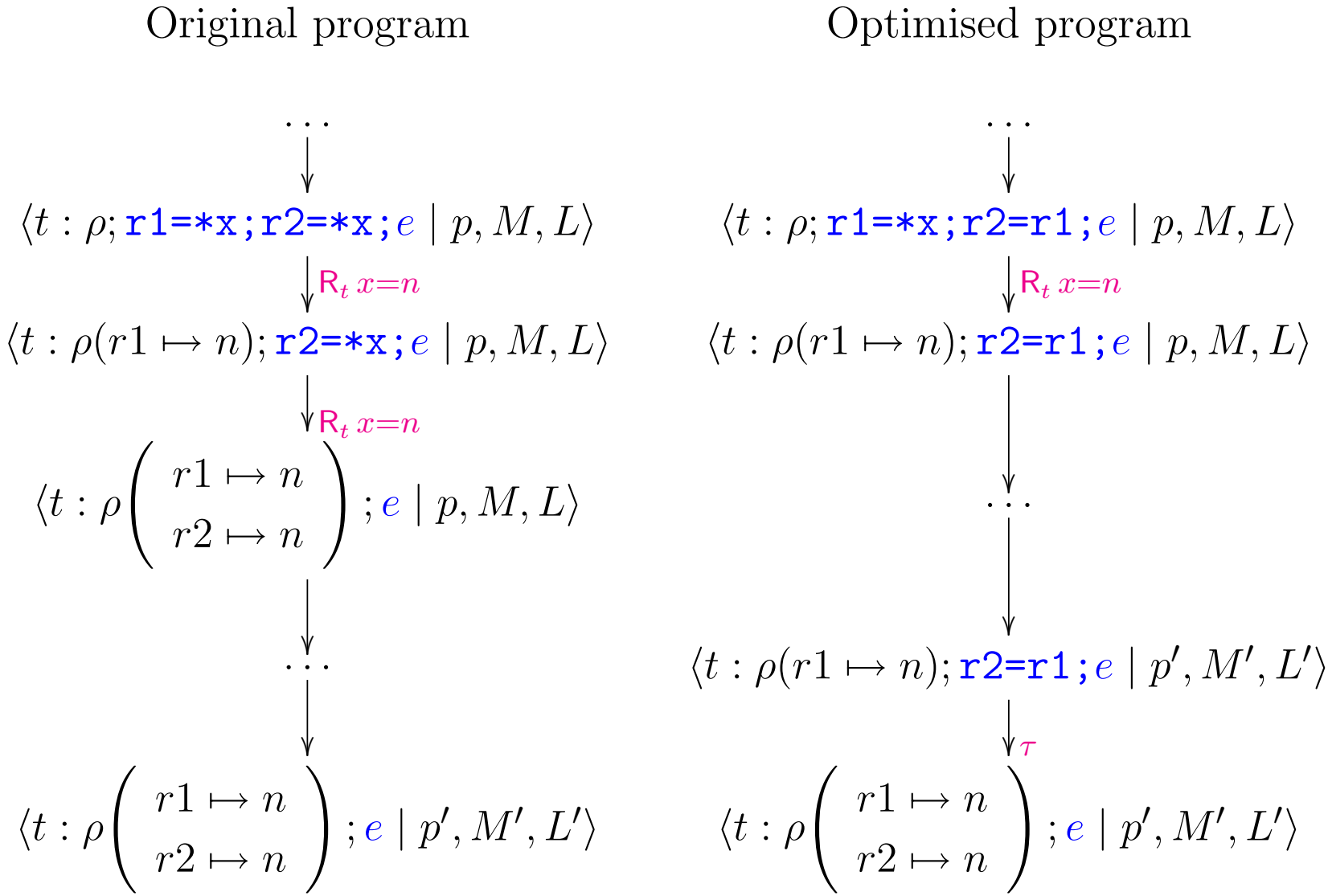
Original program



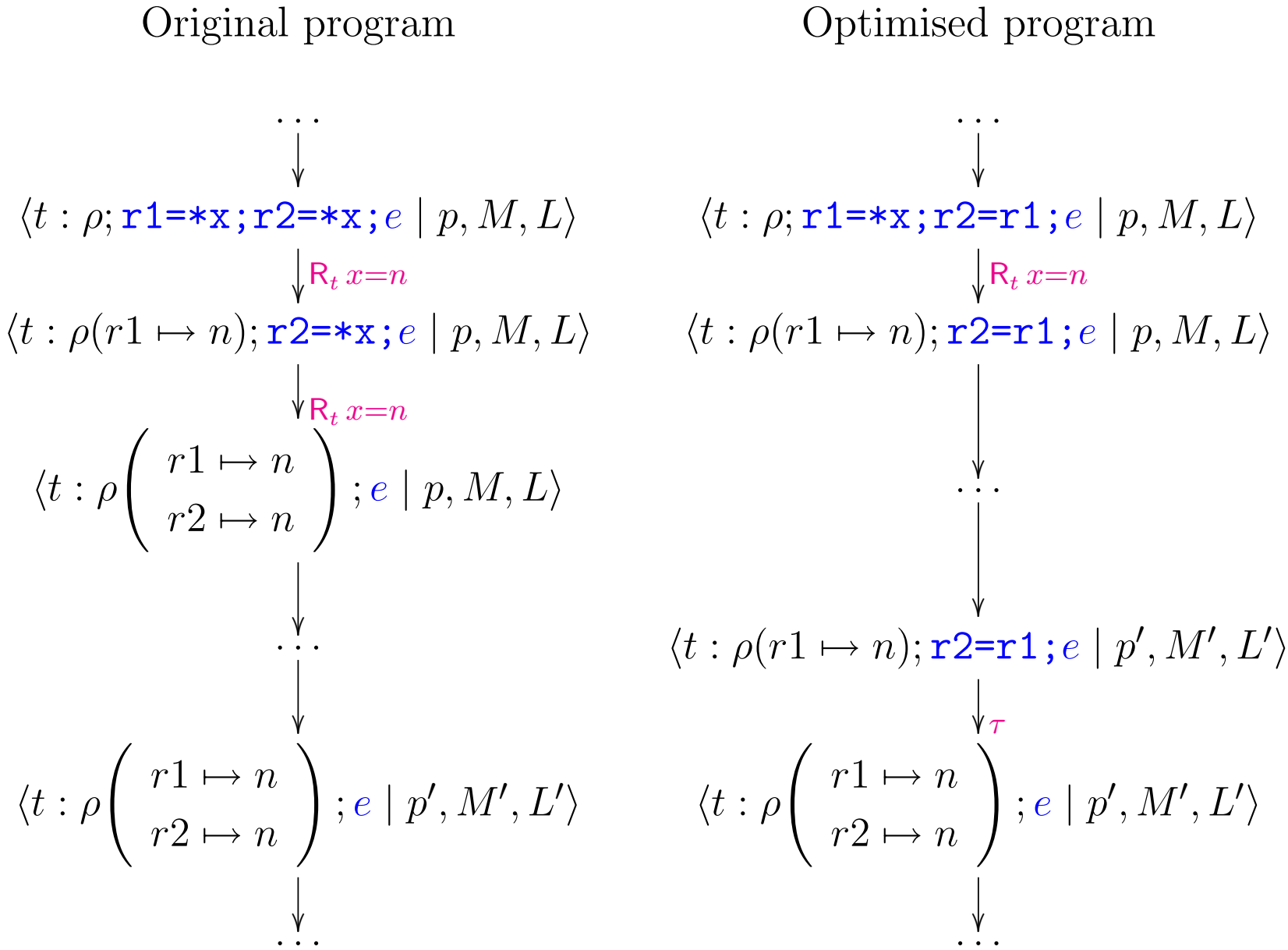
Optimised program



Simulating Elimination



Simulating Elimination



Note: there is a bit of cheating w.r.t. the semantics.

Elimination Formally

$\boxed{e_1 \rightsquigarrow e_2}$ e_2 is an elimination of e_1

$$\frac{}{r_1 = *x; r_2 = *x \rightsquigarrow r_1 = *x; r_2 = r_1}\ \textsc{Eelim}$$

$$\frac{}{e \rightsquigarrow e}\ \textsc{Eid}$$

$$\frac{e \rightsquigarrow e'}{r = e \rightsquigarrow r = e'}\ \textsc{Eassign\_context}$$

$$\frac{e \rightsquigarrow e'}{*x = e \rightsquigarrow *x = e'}\ \textsc{Ewrite\_context}$$

$$\frac{\begin{array}{c} e_1 \rightsquigarrow e_1' \\ e_2 \rightsquigarrow e_2' \end{array}}{e_1; e_2 \rightsquigarrow e_1'; e_2'}\ \textsc{Eseq\_context}$$

$$\frac{e \rightsquigarrow e'}{\mathbf{print}\ e \rightsquigarrow \mathbf{print}\ e'}\ \textsc{Eprint\_context}$$

Elimination Formally

$$\frac{\begin{array}{c} e_1 \rightsquigarrow e_1' \\ e_2 \rightsquigarrow e_2' \\ e_3 \rightsquigarrow e_3' \\ e_4 \rightsquigarrow e_4' \end{array}}{\textbf{if } e_1 = e_2 \textbf{ then } e_3 \textbf{ else } e_4 \rightsquigarrow \textbf{if } e_1' = e_2' \textbf{ then } e_3' \textbf{ else } e_4'} \quad \textsc{Eif\_context}$$

$\boxed{p_1 \rightsquigarrow p_2}$ p_2 is an elimination of p_1

$$\frac{e \rightsquigarrow e'}{t{:}\rho; e \rightsquigarrow t{:}\rho; e'} \quad \textsc{Ethread}$$

$$\frac{\begin{array}{c} p_1 \rightsquigarrow p_1' \\ p_2 \rightsquigarrow p_2' \end{array}}{p_1 | p_2 \rightsquigarrow p_1' | p_2'} \quad \textsc{Epar}$$

Proof

We want to show that for every trace of the optimised program, there is a trace of the original program with the same behaviour.

What do we mean by the trace of program $\langle p,\ M,\ L \rangle$, exactly?

Proof

We want to show that for every trace of the optimised program, there is a trace of the original program with the same behaviour.

What do we mean by the trace of program $\langle p,\ M,\ L\rangle$, exactly?

We write $\langle p,\ M,\ L\rangle \xrightarrow{t} \langle p', M', L'\rangle$ if

$$\langle p,\ M,\ L\rangle \xrightarrow{l_1} \langle p_1, M_1, L_1\rangle \xrightarrow{l_2} \cdots \xrightarrow{l_n} \langle p', M', L'\rangle$$

for some p_i, M_i, L_i and l_i such that $t = [l_1, \ldots, l_n]$.

Proof

We want to show that for every trace of the optimised program, there is a trace of the original program with the same behaviour.

What do we mean by the trace of program $\langle p,\ M,\ L\rangle$, exactly?

We write $\langle p,\ M,\ L\rangle \xrightarrow{t} \langle p', M', L'\rangle$ if

$$\langle p,\ M,\ L\rangle \xrightarrow{l_1} \langle p_1, M_1, L_1\rangle \xrightarrow{l_2} \cdots \xrightarrow{l_n} \langle p', M', L'\rangle$$

for some p_i, M_i, L_i and l_i such that $t = [l_1, \ldots, l_n]$.

Then we can say that $\langle p,\ M,\ L\rangle$ has trace t (written $\langle p,\ M,\ L\rangle \downarrow t$) if $\langle p,\ M,\ L\rangle \xrightarrow{t} \langle p',\ M',\ L'\rangle$ for some p', M' and L'.

Proof - failed attempt

Let $p \rightsquigarrow q$. We want to show that for any t, if $\langle q, M, L\rangle \downarrow t$, then there is t' such that $\langle p, M, L\rangle \downarrow t'$ and $P(t) = P(t')$, where $P(t)$ is the subsequence of observable events in t.

1. Strategy: by induction on the length of trace t.
2. Base case is trivial.

Proof - failed attempt

Let $p \rightsquigarrow q$. We want to show that for any t, if $\langle q, M, L\rangle \downarrow t$, then there is t' such that $\langle p, M, L\rangle \downarrow t'$ and $P(t) = P(t')$, where $P(t)$ is the subsequence of observable events in t.

1. Strategy: by induction on the length of trace t.
2. Base case is trivial.
3. Induction step is interesting: We have
   $\langle q, M, L\rangle \xrightarrow{t} \langle q', M', L'\rangle$ where $0 < |t|$.
   Suppose $t = t' +\!\!+ [l]$, so we have
   $\langle q, M, L\rangle \xrightarrow{t'} \langle \bar{q}, \bar{M}, \bar{L}\rangle \xrightarrow{l} \langle q', M', L'\rangle$ and we can use the induction hypothesis to get $\langle p, M, L\rangle \xrightarrow{t'} \langle \hat{p}, \hat{M}, \hat{L}\rangle$.

Proof - failed attempt

Let $p \rightsquigarrow q$. We want to show that for any t, if $\langle q, M, L\rangle \downarrow t$, then there is t' such that $\langle p, M, L\rangle \downarrow t'$ and $P(t) = P(t')$, where $P(t)$ is the subsequence of observable events in t.

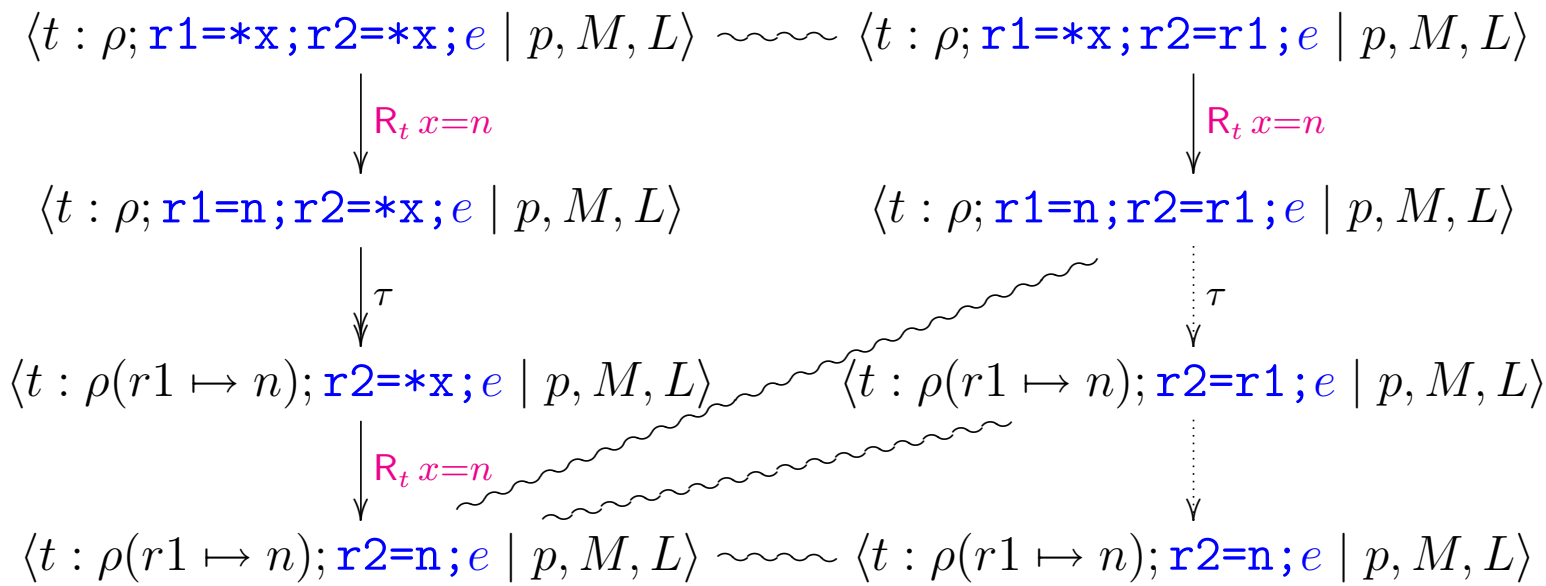
1. Strategy: by induction on the length of trace t.
2. Base case is trivial.
3. Induction step is interesting: We have
   $\langle q, M, L\rangle \xrightarrow{t} \langle q', M', L'\rangle$ where $0 < |t|$.
   Suppose $t = t' +\!\!+ [l]$, so we have
   $\langle q, M, L\rangle \xrightarrow{t'} \langle \bar{q}, \bar{M}, \bar{L}\rangle \xrightarrow{l} \langle q', M', L'\rangle$ and we can use the induction hypothesis to get $\langle p, M, L\rangle \xrightarrow{t'} \langle \hat{p}, \hat{M}, \hat{L}\rangle$.

   Uhm, this is no good because we do not know anything about $\hat{p}$, $\hat{M}$, $\hat{L}$!

Formal proof - simulation relation

Idea: we need to relate the state of the optimised and original programs so that we can simulate the step and keep being related. Our simulation relation $\sim$ must:

- Relate a program and its eliminated counterpart, i.e., if $p \rightsquigarrow q$, then $p \sim q$.
- But it must also account for the intermediate step:



Simulation relation formally

Simulation relation for elimination and intermediate states:

$$\frac{e \rightsquigarrow e'}{\rho, e \sim \rho, e'} \quad \textsc{SEElim}$$

$$\frac{}{\rho \oplus (r_1 \mapsto n), r_2 = n \sim \rho, r_1 = n; r_2 = r_1} \quad \textsc{SEAssign\_N}$$

$$\frac{}{\rho \oplus (r_1 \mapsto n), r_2 = n \sim \rho \oplus (r_1 \mapsto n), n; r_2 = r_1} \quad \textsc{SESeq}$$

$$\frac{}{\rho \oplus (r_1 \mapsto n), r_2 = n \sim \rho \oplus (r_1 \mapsto n), r_2 = r_1} \quad \textsc{SEAssign\_Reg}$$

Simulation relation formally

Simulation relation context rules:

$$\frac{\rho, e_1 \sim \rho', e_1' \quad e_2 \rightsquigarrow e_2'}{\rho, e_1; e_2 \sim \rho', e_1'; e_2'} \quad \textsc{SeSeq\_context}$$

$$\frac{\rho, e \sim \rho', e'}{\rho, \mathbf{print}\ e \sim \rho', \mathbf{print}\ e'} \quad \textsc{SePrint\_context}$$

$$\frac{\rho, e \sim \rho', e'}{\rho, r = e \sim \rho', r = e'} \quad \textsc{SeAssign\_context}$$

$$\frac{\rho, e \sim \rho', e'}{\rho, *x = e \sim \rho', *x = e'} \quad \textsc{SeWrite\_context}$$

Simulation relation formally

Simulation relation lifiting to processes:

$\boxed{p_1 \sim p_2}$ p_1 simulates p_2

$$\frac{\rho, e \sim \rho', e'}{t{:}\rho; e \sim t{:}\rho'; e'} \quad \textsc{SPthread}$$

$$\frac{\begin{array}{c} p_1 \sim p_1' \\ p_2 \sim p_2' \end{array}}{p_1 | p_2 \sim p_1' | p_2'} \quad \textsc{SPpar}$$

Simulation of expressions

First, we should make sure that a step in an optimised expression can be simulated by step in any related expression. Formally, we want to establish that if $\rho_1, e_1 \sim \rho_1', e_1'$, $\rho_1'; e_1' \xrightarrow{l} \rho_2'; e_2'$ then there are ρ_2, e_2 and trace l' such that $\rho_1; e_1 \xrightarrow{l'} \rho_2; e_2$, $\rho_2, e_2 \sim \rho_2', e_2'$, $P([l]) = P(l')$ (we should also show that l and l' have the same effect on locks/memory).

Simulation of expressions

First, we should make sure that a step in an optimised expression can be simulated by step in any related expression. Formally, we want to establish that if $\rho_1, e_1 \sim \rho_1', e_1'$, $\rho_1'; e_1' \xrightarrow{l} \rho_2'; e_2'$ then there are ρ_2, e_2 and trace l' such that $\rho_1; e_1 \xrightarrow{l'} \rho_2; e_2$, $\rho_2, e_2 \sim \rho_2', e_2'$, $P([l]) = P(l')$ (we should also show that l and l' have the same effect on locks/memory).

More graphically, we want show:

$$\begin{array}{ccc} \rho_1, e_1 & & \rho_1, e_1 \xrightarrow{l'} \rho_2, e_2 \\ \big\downarrow & \Rightarrow \exists \rho_2, e_2, l'. & \big\downarrow \qquad \big\downarrow \quad \wedge P([l]) = P(l') \\ \rho_1', e_1' \xrightarrow{l} \rho_2', e_2' & & \rho_1', e_1' \xrightarrow{l} \rho_2', e_2' \end{array}$$

Simulation of expressions

First, we should make sure that a step in an optimised expression can be simulated by step in any related expression. Formally, we want to establish that if $\rho_1, e_1 \sim \rho_1', e_1'$, $\rho_1'; e_1' \xrightarrow{l} \rho_2'; e_2'$ then there are ρ_2, e_2 and trace l' such that $\rho_1; e_1 \xrightarrow{l'} \rho_2; e_2$, $\rho_2, e_2 \sim \rho_2', e_2'$, $P([l]) = P(l')$ (we should also show that l and l' have the same effect on locks/memory).

Proof strategy: by induction on the derivation of $\rho_1'; e_1' \xrightarrow{l} \rho_2'; e_2'$ and then by case analysis on $\rho_1, e_1 \sim \rho_1', e_1'$.

Most of the induction cases are trivial. For example, if $\rho_1'; e_1' \xrightarrow{l} \rho_2'; e_2'$ was obtained by the READ rule, then $e_1' = *x$, hence $e_1 = *x$ and $\rho_1 = \rho_1'$ (by definition of $\sim$). As a result, we can take $\rho_2 = \rho_2'$, $e_2 = e_2'$ and $l' = [l]$.

Simulation of 'SEQ_CONTEXT'

If $\rho_1'; e_1' \xrightarrow{l} \rho_2'; e_2'$ was derived by SEQ_CONTEXT then $e_1' = e_1^1; e_1^2$. In this case, the case analysis on $\rho_1, e_1 \sim \rho_1', e_1'$ admits four rules: SEELIM, SEASSIGN_N and SESEQ.

Consider, for example, SEASSIGN_N. Then $e_1' = (r_1 = n; r_2 = r_1)$, $\rho_1 = \rho_1' \oplus (r_1 \mapsto n)$, $e_1 = (r_2 = n)$. Since only the CONTEXT_SEQ and VAR_WRITE rules apply to e_1', we have $\rho_2' = \rho_1' \oplus (r_1 \mapsto n)$ and $e_2' = (n; r_2 = r_1)$. Let us set $\rho_2 = \rho_1$, $e_2 = e_1$ and $l' = []$. Using SEASSIGN_SEQ, we obtain $\rho_1, e_1 \sim \rho_2', e_2'$. Since $l = \tau$, we have $P(l') = P([l])$.

Simulation of 'SEQ_CONTEXT'

If $\rho_1'; e_1' \xrightarrow{l} \rho_2'; e_2'$ was derived by SEQ_CONTEXT then $e_1' = e_1^1; e_1^2$. In this case, the case analysis on $\rho_1, e_1 \sim \rho_1', e_1'$ admits four rules: SEELIM, SEASSIGN_N and SESEQ.

Consider, for example, SEASSIGN_N. Then $e_1' = (r_1 = n; r_2 = r_1)$, $\rho_1 = \rho_1' \oplus (r_1 \mapsto n)$, $e_1 = (r_2 = n)$. Since only the CONTEXT_SEQ and VAR_WRITE rules apply to e_1', we have $\rho_2' = \rho_1' \oplus (r_1 \mapsto n)$ and $e_2' = (n; r_2 = r_1)$. Let us set $\rho_2 = \rho_1$, $e_2 = e_1$ and $l' = []$. Using SEASSIGN_SEQ, we obtain $\rho_1, e_1 \sim \rho_2', e_2'$. Since $l = \tau$, we have $P(l') = P([l])$.

The other cases are similarly tedious. You do not want to do this kind of reasoning manually! We advise to use Isabelle or HOL or Coq to check your proofs.

Proof plumbing

We want to show that for any t, if $\langle q, M, L\rangle \xrightarrow{t} \langle q', M', L'\rangle$, then there are t' and p' such that $\langle p, M, L\rangle \xrightarrow{t'} \langle p', M', L'\rangle$, $p' \sim q'$ and $P(t) = P(t')$.

Base case still trivial. Induction step:

$$\begin{array}{l} \langle p, M, L\rangle \\ \qquad \wr \\ \langle q, M, L\rangle \xrightarrow{\bar{t}} \langle \bar{q}, \bar{M}, \bar{L}\rangle \xrightarrow{l} \langle q', M', L'\rangle \end{array}$$

Use induction hypothesis!

Proof plumbing

We want to show that for any t, if $\langle q, M, L\rangle \xrightarrow{t} \langle q', M', L'\rangle$, then there are t' and p' such that $\langle p, M, L\rangle \xrightarrow{t'} \langle p', M', L'\rangle$, $p' \sim q'$ and $P(t) = P(t')$.

Base case still trivial. Induction step:

By induction hypothesis we obtain $\bar{t}'$ and $\bar{p}$ such that $\bar{p} \sim \bar{q}$, $P(\bar{t}) = P(\bar{t}')$ and

$$\begin{array}{ccccc}
\langle p, M, L\rangle & \xrightarrow{\bar{t}'} & \langle \bar{p}, \bar{M}, \bar{L}\rangle & & \\
\vdots & & \vdots & & \\
\langle q, M, L\rangle & \xrightarrow{\bar{t}} & \langle \bar{q}, \bar{M}, \bar{L}\rangle & \xrightarrow{l} & \langle q', M', L'\rangle
\end{array}$$

Proof plumbing

We need to find p' and t' such that $P(t') = P([l'])$ and

$$\begin{array}{ccc} \langle \bar{p}, \bar{M}, \bar{L} \rangle & \xrightarrow{t'} & \langle p', M', L' \rangle \\ \vdots & & \vdots \\ \langle \bar{q}, \bar{M}, \bar{L} \rangle & \xrightarrow{l} & \langle q', M', L' \rangle \end{array}$$

This is an easy proof by induction on the derivation of $\langle \bar{q}, \bar{M}, \bar{L} \rangle \xrightarrow{l} \langle q', M', L' \rangle$, using our (sketched) expression simulation lemma to do the heavy lifting.

- Compiler Optimisations
  - Correctness
- Optimisations in SC
- Optimisations of Data-race-free Programs
  - Data race freedom
  - Limitations of the DRF principle
  - Correctness under the DRF principle
- Out-of-thin-air values

Data Race Freedom Principle

Since many common optimisations are not valid in SC, language designers specify relaxed models.

Insight: to observe an optimisation one needs to be able to observe other thread's memory accesses at the same time when they are performed — data race!

So, in data race free programs, common optimisations are not observable.

DRF principle: each behaviour of a data race free program (after optimisation) must be the same as the behaviour of some (SC) execution of the program (before optimisation).

Says nothing about programs with data races!

- Compiler Optimisations
  - Correctness
- Optimisations in SC
- Optimisations of Data-race-free Programs
  - Data race freedom
  - Limitations of the DRF principle
  - Correctness under the DRF principle
- Out-of-thin-air values

What is DRF, exactly?

There is no execution with adjacent conflicting memory accesses from different threads, where a pair of memory accesses is conflicting if they access the same memory location and at least one of them is a write.

Program with a data race:

`*y = 1`	`if *x = 1`
`*x = 1`	`then print *y`

A racy execution:

$$\mathsf{W}_{t_1}\, y{=}1, \mathsf{W}_{t_1}\, x{=}1, \mathsf{R}_{t_2}\, x{=}1, \mathsf{R}_{t_2}\, y{=}1, \mathsf{P}_{t_2}\, 1$$

What is DRF, exactly?

There is no execution with adjacent conflicting memory accesses from different threads, where a pair of memory accesses is conflicting if they access the same memory location and at least one of them is a write.

Data race free program:

```
*y = 1      |  lock x;
lock x      |  r = *x;
*x = 1      |  unlock x;
unlock x    |  if r = 1
            |  then print *y
```

What is DRF, exactly?

There is no execution with adjacent conflicting memory accesses from different threads, where a pair of memory accesses is conflicting if they access the same memory location and at least one of them is a write.

Data race free program:

```
if *x = 1      |  if *y = 1
then *y = 1    |  then *x = 1
```

Note that the writes cannot be executed in any SC execution! So they cannot participate in a data race. . .

Another DRF definition

In literature, you will often find a different different definition of data race freedom — using happens-before relation.

The intuition:

- happens-before relates events that are ordered in time by synchronisation or by sequencing in a thread.
- a data race is a pair of conflicting memory accesses that is not ordered by happens-before.
- a pair of memory accesses is conflicting if they access the same memory location and at least one of them is a write.

Happens-before order

Suppose that t is a trace $t = [l_1, \ldots, l_n]$.

Let $1 \leq i, j \leq n$. We say that

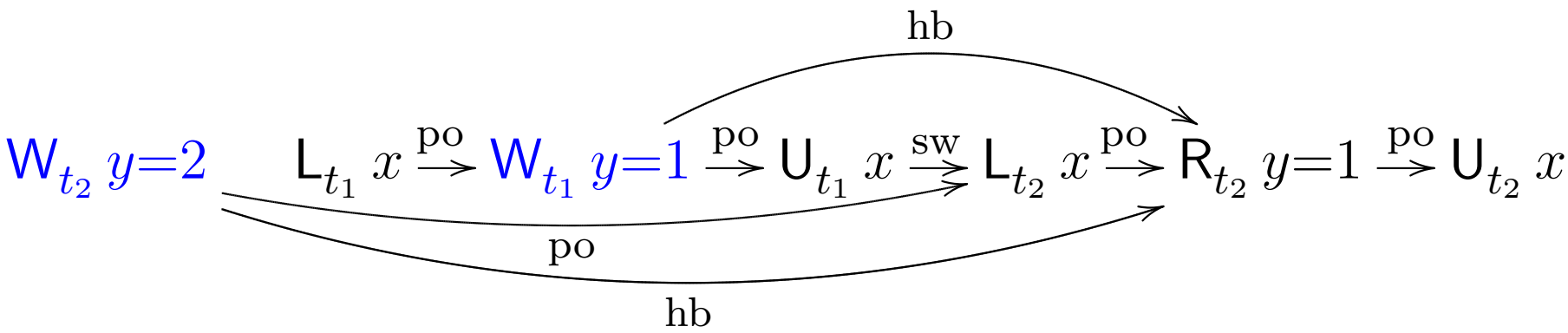
- i synchronizes-with j if $i < j$, $l_i = \mathsf{U}_{t_i}\, x$ and $l_j = \mathsf{L}_{t_j}\, x$.
- i is before j in program order if $i < j$ and l_i, l_j belong to the same thread.
- i happens-before j if $\langle i, j \rangle$ is in the transitive closure of the union of the synchronizes-with and program orders.

Happens-before order

Suppose that t is a trace $t = [l_1, \ldots, l_n]$.

Let $1 \leq i, j \leq n$. We say that

- i synchronizes-with j if $i < j$, $l_i = \mathsf{U}_{t_i}\, x$ and $l_j = \mathsf{L}_{t_j}\, x$.
- i is before j in program order if $i < j$ and l_i, l_j belong to the same thread.
- i happens-before j if $\langle i, j \rangle$ is in the transitive closure of the union of the synchronizes-with and program orders.



DRF via happens-before

We say that program is hb-data-race-free if in all executions, all different pairs of conflicting actions in the executions are ordered by the happens-before order. Here, actions are conflicting if they are memory accesses to the same location and at least one of them is a write.

Theorem (cf. Boehm, Adve, PLDI 2008): An execution is hb-data-race-free if and only if it is data-race-free.

This a very useful theorem to prove correctness of optimisations under the DRF principle.

- Compiler Optimisations
  - Correctness
- Optimisations in SC
- Optimisations of Data-race-free Programs
  - Data race freedom
  - Limitations of the DRF principle
  - Correctness under the DRF principle
- Out-of-thin-air values

Limitations of the DRF

The DRF principle allows most optimisations, but not all.

The dangerous “optimisations” are usually the ones that introduce shared-memory accesses (this is very rare):

- Write introducing code transformations
  - often speculatively, or
  - writes that would be redundant in sequential code.
- Read introducing optimisations
  - often harmless by themselves, but can be dangerous in combination with optimisations that are correct under DRF.

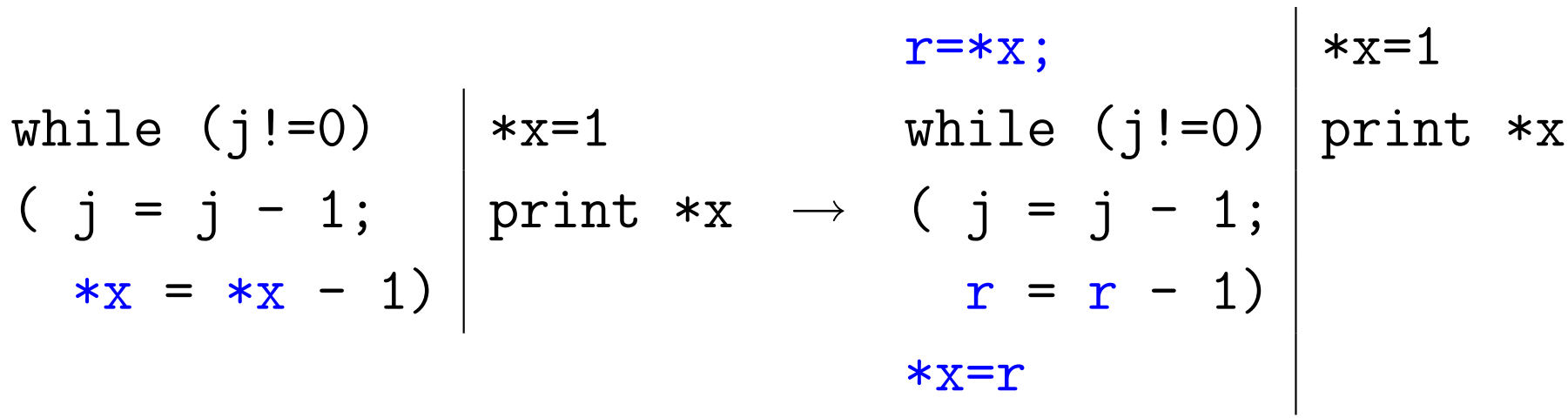
Loop variable register promotion

Is the following transformation valid (in any context)?

```
while (j != 0) (                    r = *x;
  j = j - 1;              →         while (j != 0) (
  *x = *x - 1 )                       j = j - 1;
                                      r = r - 1)
                                    *x = r
```

Loop variable register promotion

Assume the initial state `*x` $= 0$.



Can the process print 0?

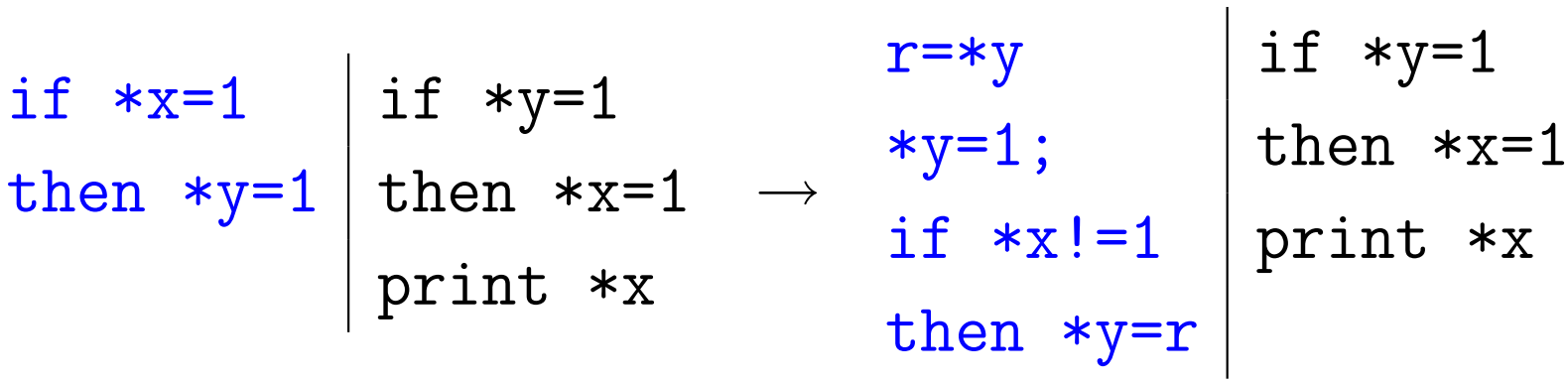
Aggressive write speculation

Is the following transformation valid (in any context)?

```
if *x = 1                    r = *y
then *y = 1        →         *y = 1;
                             if *x != 1
                             then *y = r
```

Aggressive write speculation

Assume initial state $*x = *y = 0$.



Can the program print 1?

Read introduction?

It is correct to introduce a read and then throw away its value.

But. . .

Read introduction?

It is correct to introduce a read and then throw away its value.

But. . .

Observe that it is correct (under DRF) to do CSE across `lock` (or `unlock` but not both):

`r1 = *x; lock m; r2 = *x` $\longrightarrow$ `r1 = *x; lock m; r2 = r1`

Read introduction?

It is correct to introduce a read and then throw away its value.

But. . .

Consider the following program and assume that initially `*x` $=$ `*y` $= 0$.

`lock m;`	`lock m;`
`*x = 1;`	`*y = 1;`
`print *y`	`print *x`
`unlock m`	`unlock m`

Can the program print two zeros?

Read introduction?

It is correct to introduce a read and then throw away its value.

But. . .

Consider the following program and assume that initially $*x = *y = 0$.

`r1=*y;`	`r2=*x;`
`lock m;`	`lock m;`
`*x = 1;`	`*y = 1;`
`print *y`	`print *x`
`unlock m`	`unlock m`

Can the program print two zeros?

Read introduction?

It is correct to introduce a read and then throw away its value.

But. . .

Consider the following program and assume that initially `*x` $=$ `*y` $= 0$.

`r1=*y;`	`r2=*x;`
`lock m;`	`lock m;`
`*x = 1;`	`*y = 1;`
`print *y`	`print *x`
`unlock m`	`unlock m`

Can the program print two zeros?

Read introduction?

It is correct to introduce a read and then throw away its value.

But. . .

Consider the following program and assume that initially $*x = *y = 0$.

`r1=*y;`	`r2=*x;`
`lock m;`	`lock m;`
`*x = 1;`	`*y = 1;`
`print r1`	`print r2`
`unlock m`	`unlock m`

Now the program can print two zeros.

Practical limitations of DRF principle

Data race detection is hard!

- There are type systems for DRF, but they are restrictive. . .
- There are static analyses for DRF, but they are imprecise and they must see whole program. . .
- There are run-time data race detectors, but they are slow and incomplete (they cannot guarantee data race freedom). . .