The Concurrent Programming Abstraction

Contiene illustrazioni prese da

M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

Programmazione concorrente

- Un programma concorrente contiene due o più processi (o sottoprocessi – thread) che lavorano assieme per eseguire una determinata applicazione
- Ciascun (sotto)processo è un programma sequenziale
- I (sotto)processi comunicano tra loro utilizzando variabili condivise (shared memory) o scambiandosi messaggi (message passing)
- ASTRAZIONE: i (sotto)processi sono in esecuzione contemporanea
  - E' una astrazione: in realtà, potrebbe non essere esattamente così

Le origini

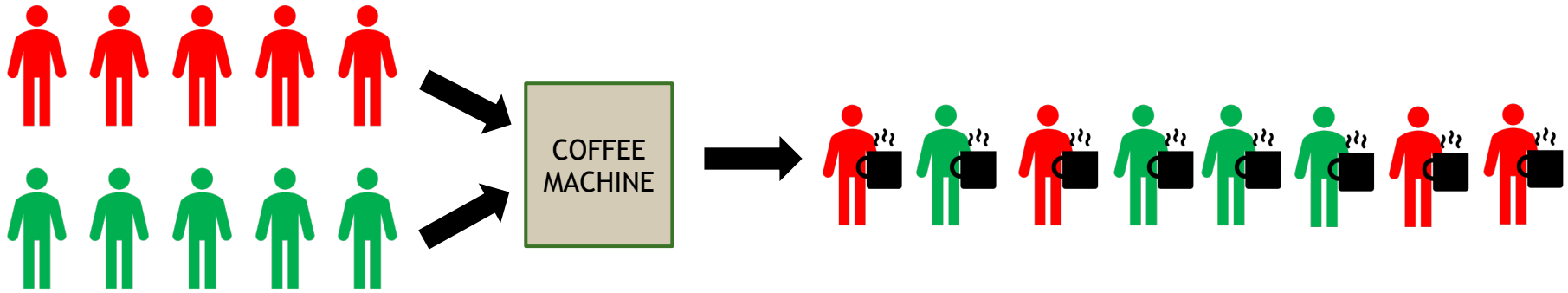
- La programmazione nasce negli anni '60 nell'ambito dei sistemi operativi
- Nasce dall'esigenza di far continuare l'esecuzione dei programmi durante lo svolgimento di (lunghe) operazioni di I/O
- Negli anni '80 si diffondono sistemi operativi con pre-emptive multitasking
  - Possibilità di mantenere attivi più programmi (o processi) contemporaneamente alternandone l'esecuzione nel processore (interleaving)
  - L'alternarsi dei programmi è sotto il controllo del sistema operativo
  - Richiede supporto hardware (interrupt programmabili)

Sistemi multiprocessore

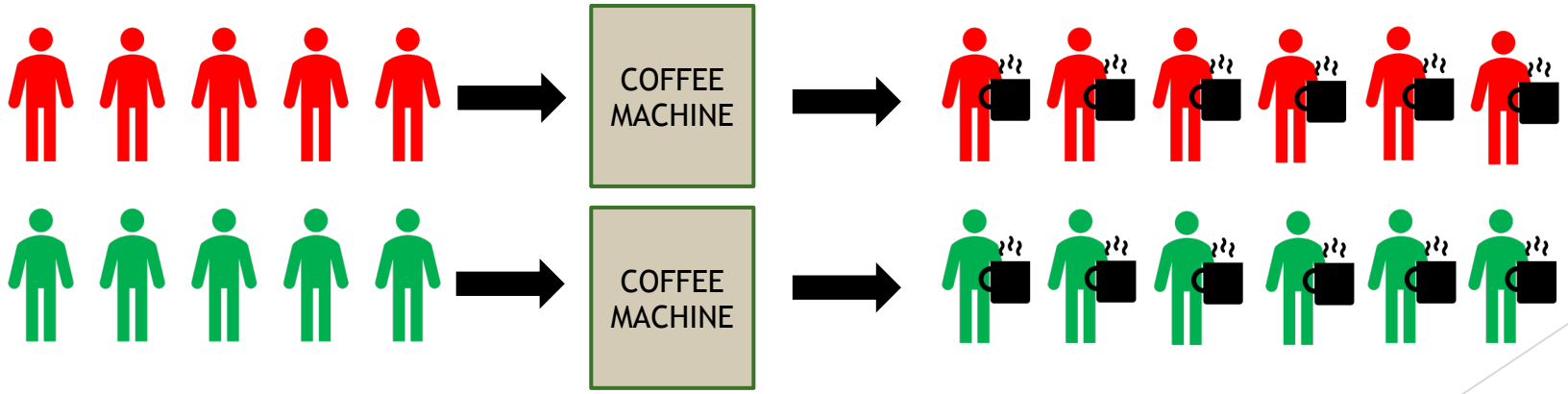
- Successivamente furono introdotti i sistemi multiprocessore, dotati di più CPU (e/o di CPU multi-core)
- I sistemi multiprocessore
  - consentono di eseguire diversi processi in parallelo (ossia, contemporaneamente su CPU diverse)
  - consentono di eseguire le applicazioni (costituite da più processi) più velocemente rispetto all'esecuzione su singola CPU

Concorrenza VS Parallelismo

- Programmazione concorrente: 1 CPU e N task contemporaneamente



- Programmazione parallela: N CPU e N task contemporaneamente



Ma frequentemente capita di avere N CPU e M task con N<M

Esecuzione non sequenziale

- Concorrenza e parallelismo sono accomunati da un aspetto
  - Esecuzione non sequenziale del programma
  - Idealmente, ogni (sotto)processo che costituisce il programma ha un proprio program counter che avanza autonomamente
- Esempio con due processi, ognuno con due comandi
  - Processo P: p1,p2
  - Processo Q: q1,q2

- Possibili interleaving in caso di concorrenza (singola CPU)

  p1 -> p2 -> q1 -> q2

  p1 -> q1 -> p2 -> q2

  p1 -> q1 -> q2 -> p2

  q1 -> p1 -> p2 -> q2

  q1 -> p1 -> q2 -< p2

  q1 -> q2 -> p1 -> p2

Esempio

Algorithm 2.1: Trivial concurrent program	
integer n ← 0	
p	**q**
integer k1 ← 1 p1: n ← k1	integer k2 ← 2 q1: n ← k2

- n è una variabile globale (inizializzata a 0)
- k1 e k2 sono variabili locali
- p e q sono processi concorrenti

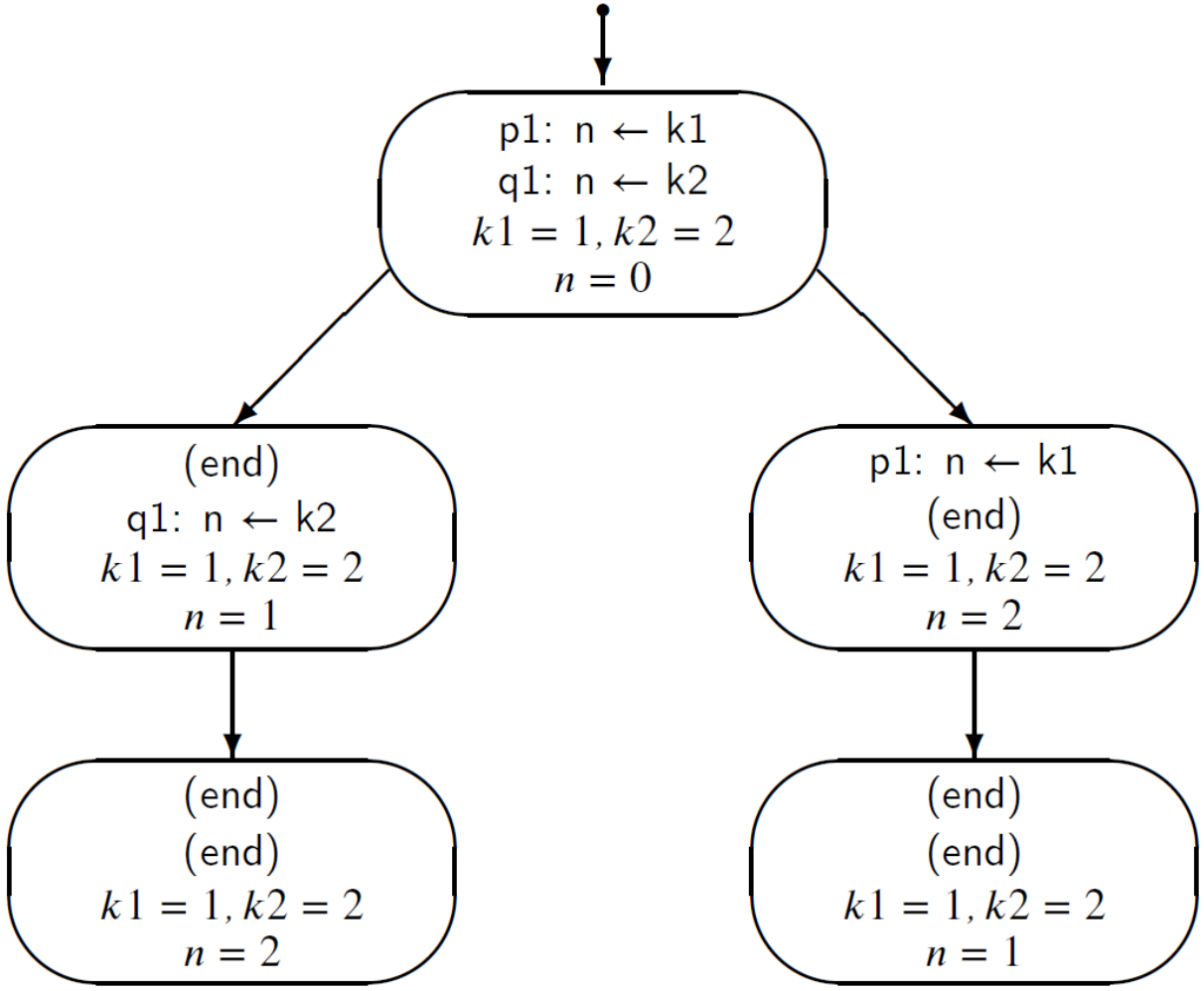
qual è il valore di n al termine?

Analizziamo i possibili comportamenti dell'esempio tramite un sistema di transizioni

- E' dato dalla semantica del linguaggio
- Descrive tutti i possibili passi di computazione che il programma può fare
- Tiene conto dei possibili interleaving dei processi

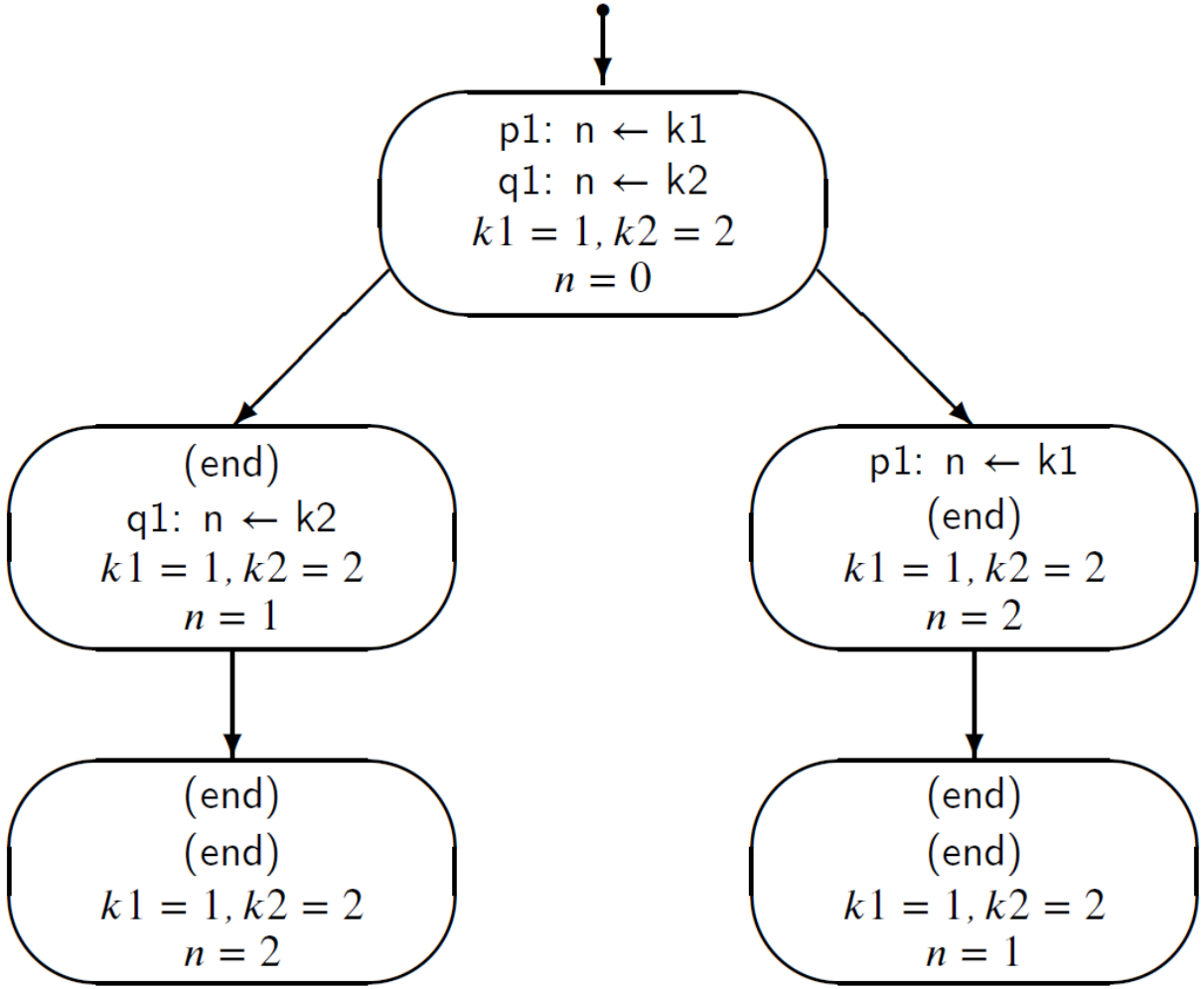
Esempio:

- n ha due possibili valori finali: 2 e 1



Il sistema di transizioni

- E' non deterministico (lo stato iniziale ha due alternative possibile)
- Il non determinismo ASTRAE dal criterio usato dallo scheduler del sistema operativo per scegliere quale processo far avanzare (es. processi prioritari su altri)
- Conoscendo lo scheduler si potrebbe concludere che alcuni cammini nel sistema di transizioni non sono in realtà realizzabili
- Il sistema di transizioni non fa assunzioni sullo scheduler (descrive tutte le esecuzioni possibili)



M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

Operazioni atomiche

Algorithm 2.1: Trivial concurrent program	
integer n ← 0	
p	**q**
integer k1 ← 1 p1: n ← k1	integer k2 ← 2 q1: n ← k2

L'analisi che abbiamo fatto è corretta sotto l'assunzione che le operazioni di assegnamento p1 e q1 siano atomiche

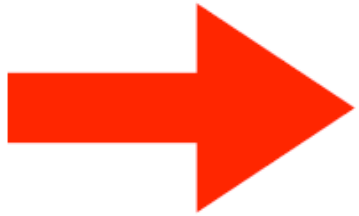
- elementari e non interrompibili

Ma non abbiamo tenuto conto del compilatore!

Il compilatore

Il compilatore traduce un singolo comando del linguaggio di alto livello in una sequenza di operazioni nel linguaggio macchina

```
n = k + 1;
```



```
load  R1,k
add   R1,#1
store R1,n
```

nota: R1 è un registro

Analisi al livello macchina

In pratica, l'interleaving ha luogo al livello del linguaggio macchina e non del linguaggio di alto livello

- Sono le operazioni assembler ad alternarsi, non i comandi del linguaggio di alto livello

Analisi al livello macchina

Dovremmo rivedere l'analisi dell'esempio di programma concorrente considerandolo così

Algorithm 2.1: Trivial concurrent program

integer n ← 0

p	q
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

p1: load R1,k1
p2: store R1,n

q1: load R1,k2
q2: store R1,n

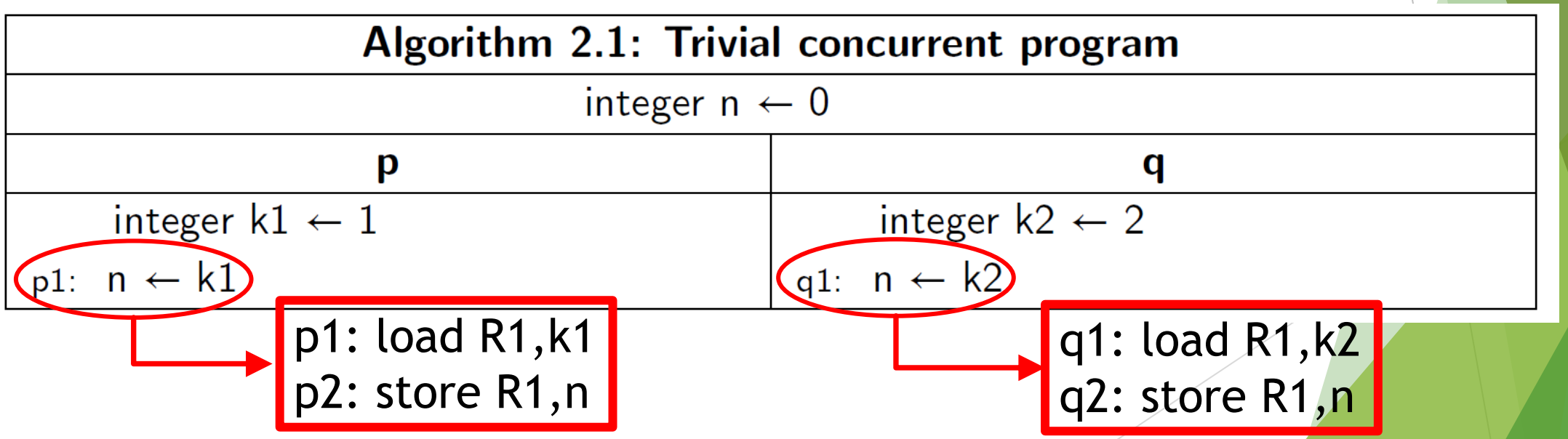
Immaginiamo che ogni processo abbia un proprio registro R1 (vedrete a S.O. come questo è reso possibile)

TUTTO OK! Come prima, deduciamo che n alla fine potrebbe avere valore 1 o 2 (a seconda che l'ultima operazione eseguita sia p2 o q2)

E in caso di parallelismo?

Altro problema: se p e q fossero eseguiti su due CPU diverse, le operazioni p2 e q2 potrebbero essere eseguite contemporaneamente

- Non in interleaving, ma in "true parallelism"



E in caso di parallelismo?

Altro problema: se p e q fossero eseguiti su due CPU diverse, le operazioni p2 e q2 potrebbero essere eseguite contemporaneamente

- Non in interle

Che succede se due processi cercano di scrivere contemporaneamente la stessa variabile globale?

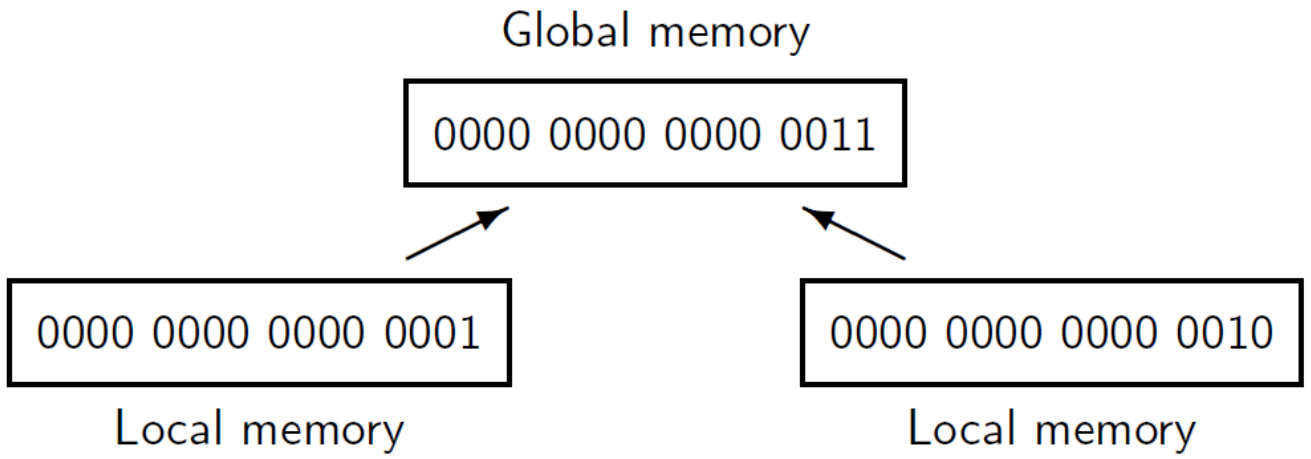
A	
	integer n ← 0
p	**q**
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

p1: load R1,k1
p2: store R1,n

q1: load R1,k2
q2: store R1,n

Scenario possibile?

Modificando contemporaneamente la stessa locazione di memoria i due processi potrebbero sovrapporre i propri risultati?



Scenario possibile?

Modificando contemporaneamente la stessa locazione di memoria i due processi potrebbero sovrapporre i propri risultati?

NO!

Anche in caso di parallelismo l'hardware garantisce che al più un processo per volta possa scrivere una certa locazione di memoria

Quindi…

Il fatto che anche in un contesto di parallelismo le modifiche contemporanee alla stessa locazione di memoria siano gestite dall'hardware eseguendole una per volta ci riporta in uno scenario di concorrenza

- I processi possono operare in parallelo solo su locazioni di memoria distinte

- La concorrenza (interleaving) è un'astrazione che consente di fare molte analisi dei programmi che varranno anche in situazioni di parallelismo
- Come per lo scheduling del sistema operativo, anche le scelte operate dall'hardware saranno modellate tramite non determinismo nei sistemi di transizione

Analisi a livello di macchina VS Analisi ad alto livello

Torniamo sulla questione dell'analisi a livello macchina con un altro esempio

Algorithm 2.3: Atomic assignment statements	
integer n ← 0	
p	**q**
p1: n ← n + 1	q1: n ← n + 1

Ragioniamo ad alto livello, quindi assumiamo che gli assegnamenti (del linguaggio di programmazione) siano operazioni atomiche (indivisibili)

Che valore ha n alla fine?

Analisi a livello di macchina VS Analisi ad alto livello

Queste sono le due (uniche) esecuzioni possibili del nuovo esempio di programma concorrente

- Corrispondono a due cammini alternativi nel sistema di transizioni che descrive il comportamento del programma
- In entrambi i casi alla fine n vale 2

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	**q1: n←n+1**	1
(end)	(end)	2

Process p	Process q	n
p1: n←n+1	**q1: n←n+1**	0
p1: n←n+1	(end)	1
(end)	(end)	2

Analisi a livello di macchina VS Analisi ad alto livello

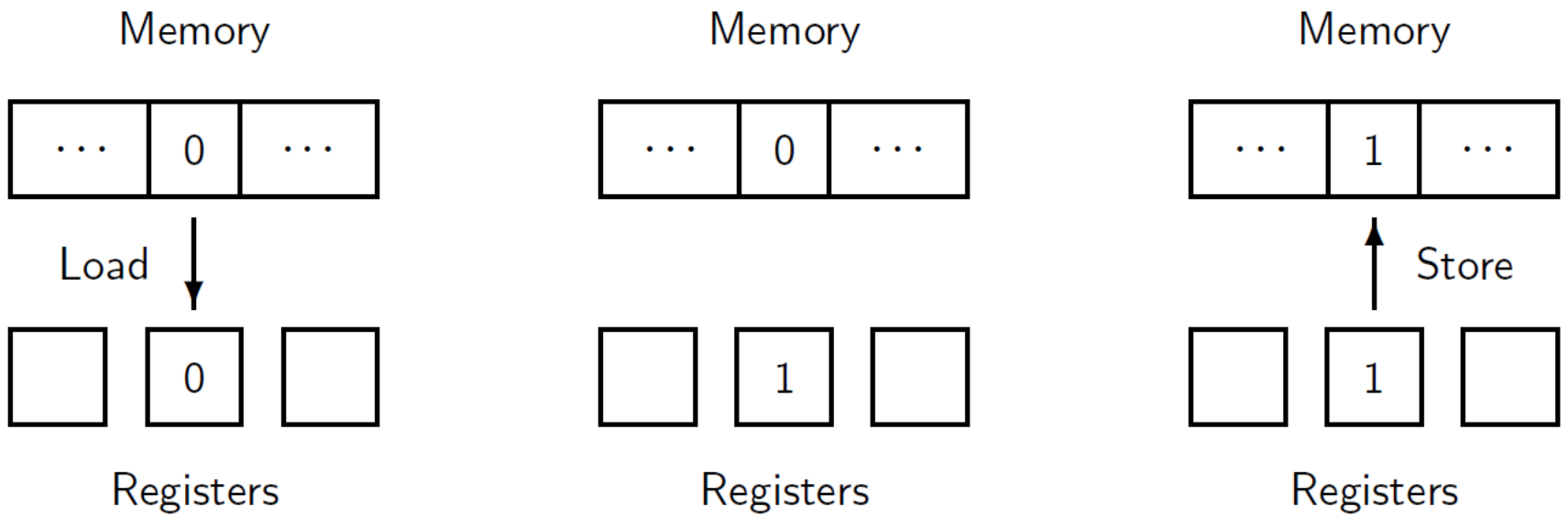
Vediamo ora lo stesso programma a livello macchina

- Queste le corrispondenti istruzioni in assembler

Algorithm 2.6: Assignment statement for a register machine	
integer n ← 0	
p	**q**
p1: load R1,n	q1: load R1,n
p2: add R1,#1	q2: add R1,#1
p3: store R1,n	q3: store R1,n

Analisi a livello di macchina VS Analisi ad alto livello

- Rappresentazione schematica dell'esecuzione di ognuno dei due processi



Analisi a livello di macchina VS Analisi ad alto livello

Questa è una possibile esecuzione descritta al livello di linguaggio macchina

- in questa particolare esecuzione alla fine n vale 1…
- Rispetto a prima, la load di un processo non è successiva alla store dell'altro
- chiaramente ci sono altre esecuzioni possibili in cui n alla fine vale 2 (ad es. p1,p2,p3,q1,q2,q3)
- Questa esecuzione non è colta dalla descrizione ad alto livello del comportamento (la semantica del linguaggio di programmazione)
- L'analisi ad alto livello non è completa (non cattura tutti i comportamenti possibili)

Process p	Process q	n	p.R1	q.R1
p1: load R1,n	q1: load R1,n	0	?	?
p2: add R1,#1	**q1: load R1,n**	0	0	?
p2: add R1,#1	q2: add R1,#1	0	0	0
p3: store R1,n	**q2: add R1,#1**	0	1	0
p3: store R1,n	q3: store R1,n	0	1	1
(end)	**q3: store R1,n**	1	1	1
(end)	(end)	1	1	1

Simulare il comportamento a livello macchina nel linguaggio ad alto livello

Possibile soluzione

- simulare l'interleaving che si ha a livello macchina con assegnamenti a variabili temporanee (locali) nel linguaggio ad alto livello per disaccoppiare load e store alle variabili globali (che sono condivise tra processi)

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Nota: p e q usano due variabili temp diverse

Simulare il comportamento a livello macchina nel linguaggio ad alto livello

Questo consente di catturare ad alto livello anche il comportamento di basso livello che nel caso precedente non veniva descritto

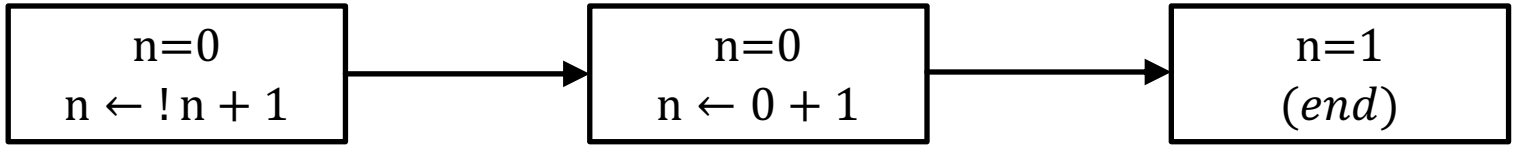
Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	**q1: temp←n**	0	0	?
p2: n←temp+1	q2: n←temp+1	0	0	0
(end)	**q2: n←temp+1**	1	0	0
(end)	(end)	1	0	0

Nota: p e q usano due variabili temp diverse

Simulare il comportamento a livello macchina nel linguaggio ad alto livello

Altro modo di effettuare la stessa simulazione del comportamento di basso livello consiste nell'assumere nel linguaggio un'operazione di dereferenziazione esplicita delle variabili

- Invece di scrivere $\mathbf{n \leftarrow n + 1}$ scriveremo $\mathbf{n \leftarrow !\, n + 1}$ dove **!** rende esplicita l'operazione di lettura della variabile (richiede un passo di computazione)
- Esecuzione:



è una soluzione meno realistica (il valore 0 letto dalla variabile viene temporaneamente scritto nel codice del programma…) ma che cattura correttamente quello che accade a livello più basso (lettura e scrittura in due passi)

Ricapitolando

Possiamo usare la concorrenza come astrazione che

1. descrive il comportamento di processi tramite interleaving
2. formalizzandoli come sistemi di transizioni non deterministici
3. ottenuti dalla semantica del linguaggio di alto livello

Questa astrazione

- Cattura anche aspetti di parallelismo
  - in particolare sulle variabili globali condivise
- Cattura anche aspetti di interleaving a livello macchina
  - eventualmente aggiungendo variabili temporanee locali o esplicitando le operazioni di dereferenziazione

Shared Memory VS Message Passing

Shared memory VS Message Passing

Per coordinare il lavoro del programma, è frequente che due o più (sotto)processi debbano comunicare tra loro

- Per sincronizzarsi in certi momenti
- Per scambiarsi dati

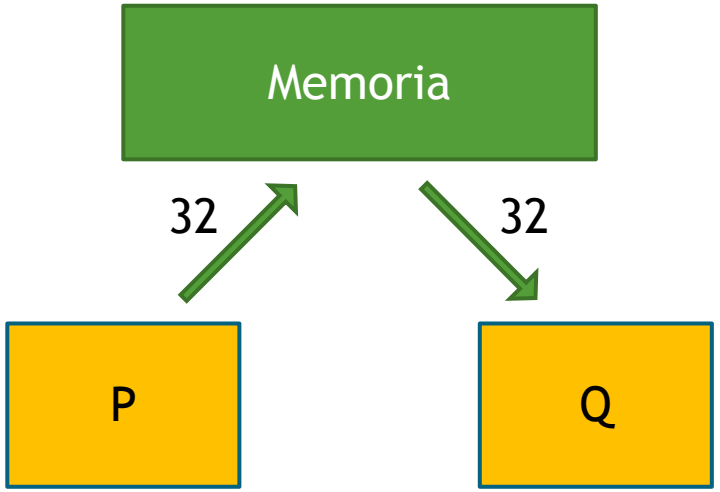
Due modi diversi per far comunicare processi

- Memoria condivisa (shared memory)
- Scambio di messaggi (message passing)

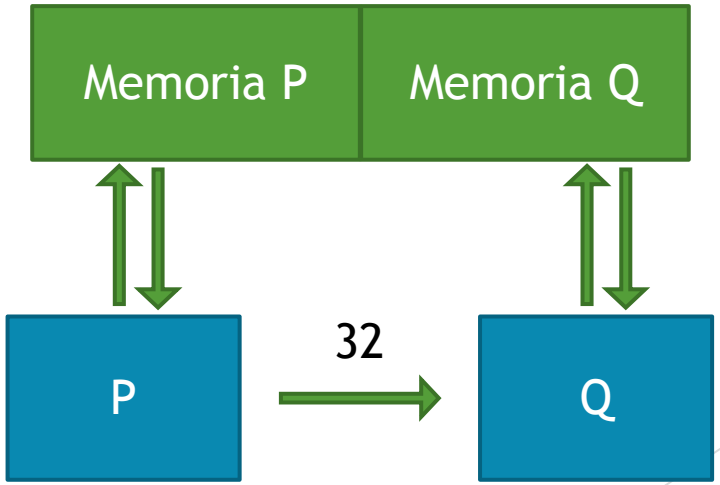
Shared Memory VS Message Passing

Shared Memory

- i processi (o thread, in questo caso) possono accedere alle stesse aree di memoria
- si sincronizzano e comunicano tra loro scrivendo e leggendo variabili condivise



Shared Memory

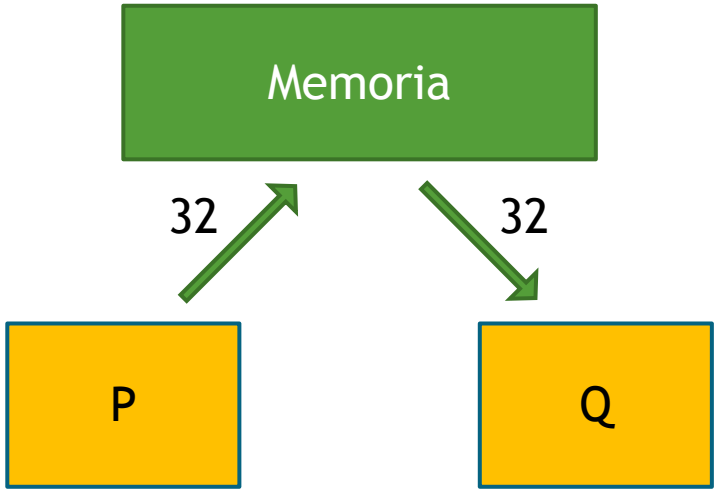


Message Passing

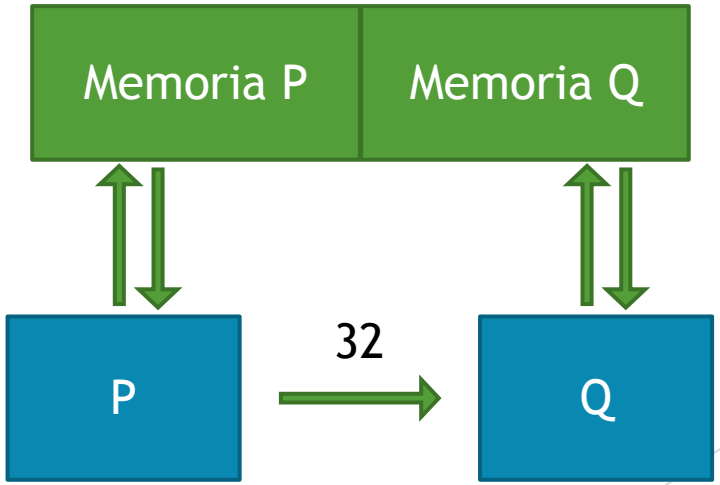
Shared Memory VS Message Passing

Message Passing

- i processi accedono ad aree diverse della memoria, ma possono inviarsi messaggi e sincronizzarsi usando servizi di inter-process communication (IPC) messi a disposizione dal sistema operativo



Shared Memory



Message Passing

Esempio di comunicazione con shared memory (in pseudocodice)

```
// variabili globali (memoria condivisa)
int x = 0;

// THREAD 1                    // THREAD 2
producer() {                   consumer() {
  int k = 6;                     while (x==0) sleep(10);
  x = fattoriale(k);             print(x); // stampa 720
}                              }
```

Il thread consumer attende che il producer abbia scritto un valore in x usando una tecnica di busy waiting (testa il valore di x ogni 10 millisecondi)

- i due thread si sincronizzano! (il secondo attende il primo)
- la strategia busy waiting è inefficiente (il secondo thread consuma tempo di CPU solo per testare ripetutamente il valore di x)

Esempio di comunicazione con shared memory (in pseudocodice)

```
// variabili globali (memoria condivisa)
int x = 0;

// THREAD 1                    // THREAD 2
producer() {                   consumer() {
  int k = 6;                     if (x==0) sleep();
  x = fattoriale(k);             print(x); // stampa 720
  wakeup();                    }
}
```

Il runtime del linguaggio può mettere a disposizione servizi di segnalazione tra thread

- Chiamando sleep() il consumer si mette in attesa
- Chiamando wakeup() il producer sblocca il consumer segnalandogli che il dato è pronto per essere letto

Esempio di comunicazione con message passing (in pseudocodice)

```
// PROCESSO 1
producer() {
   int k = 6;
   int x = fattoriale(k);
   send(x);
}
```

```
// PROCESSO 2
   consumer() {
      int y = receive();
      print(y); // stampa 720
   }
```

Il sistema operativo mette a disposizione dei processi un vero e proprio servizio di messaggistica (Inter-Process Communication, IPC)

- chiamando receive() il processo consumer si mette in attesa di ricevere messaggi
- chiamando send(x) il processo producer sblocca il consumer e gli passa il valore di x
- il trasferimento del dato dalla memoria di un processo a quella dell'altro è a carico del sistema operativo

Necessità di meccanismi di sincronizzaizone

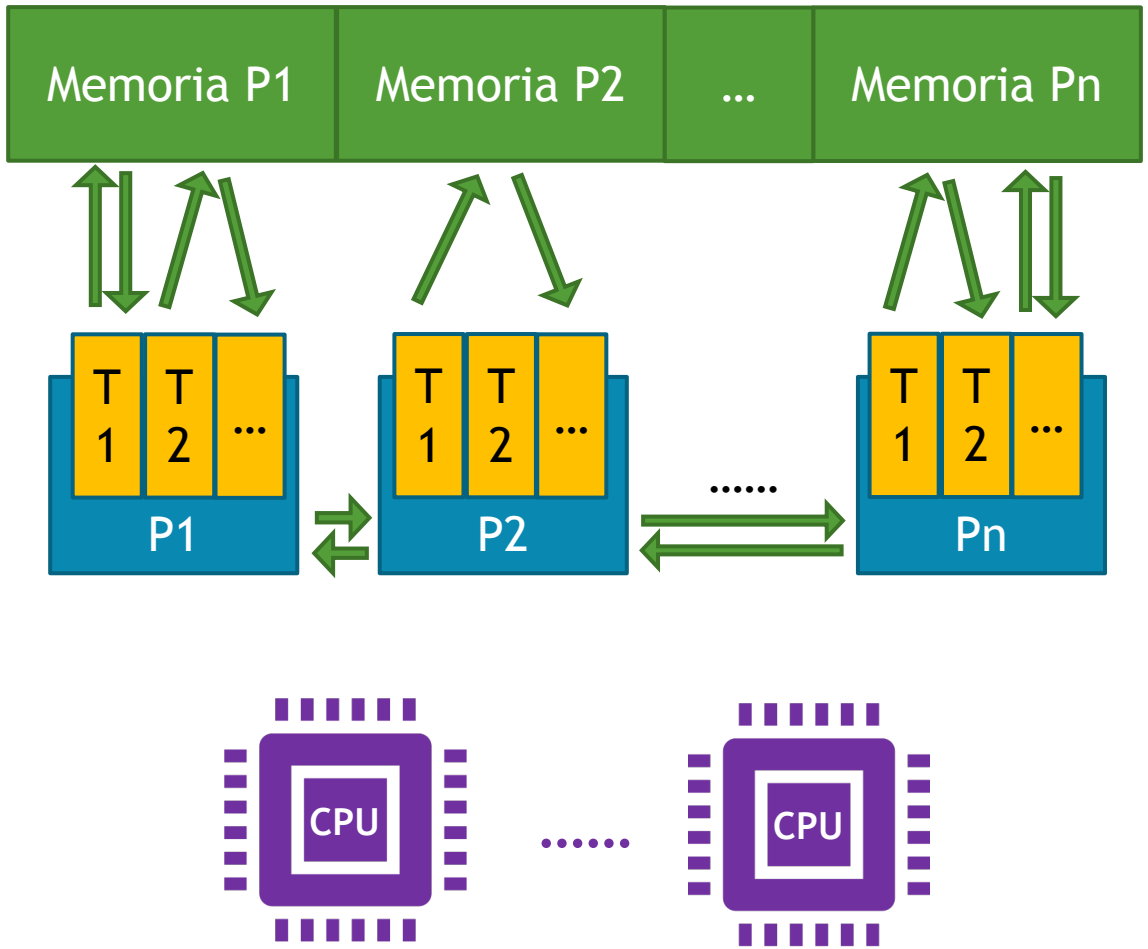
Gli esempi mostrano l'importanza di meccanismi di sincronizzazione

- busy waiting
- sleep() e wakeup()
- send() e receive()
- …

Nel caso dei thread questi meccanismi possono essere realizzati a livello di runtime del linguaggio di programmazione

Nel caso di processi, deve necessariamente essere il sistema operativo a fornire servizi di sincronizzazione e comunicazione (i processi sono isolati l'uno dall'altro)

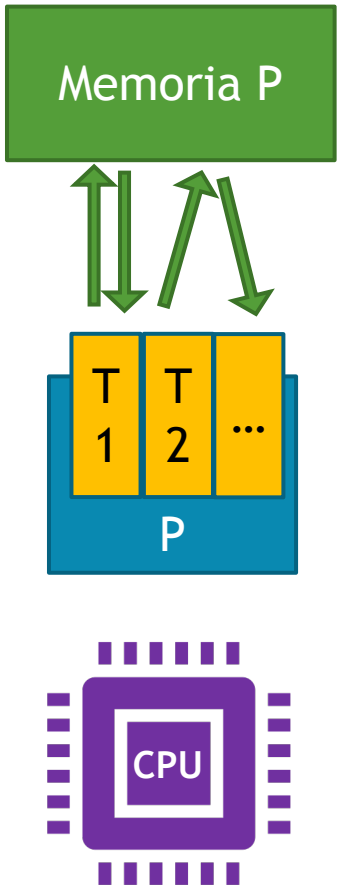
Visione astratta (e semplificata) dell'hardware e del sistema operativo



In generale

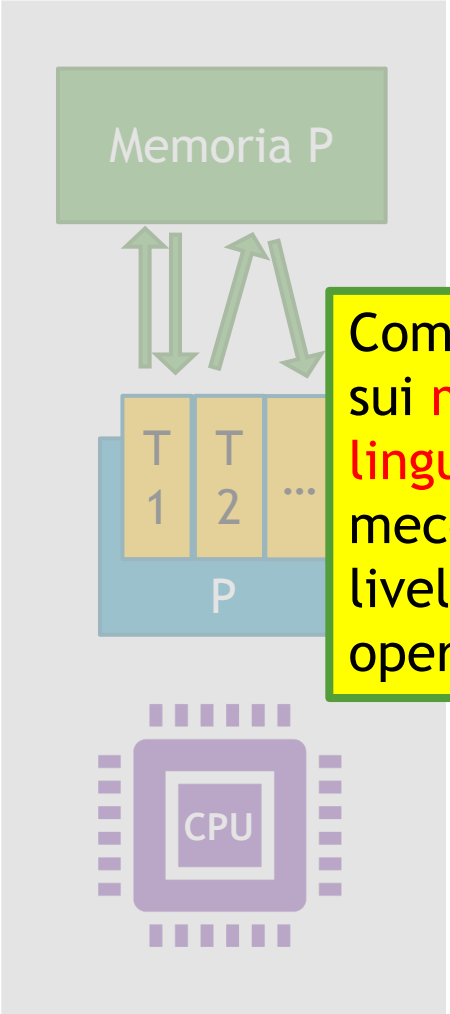
- il sistema operativo può mantenere in esecuzione vari processi fornendo servizi di IPC
- ogni processo può prevedere uno o più thread, che condividono tra loro la memoria del processo
- i processi (e i loro thread) possono essere eseguiti in parallelo su più CPU

Scenario che consideriamo per studiare la concorrenza (multithreading)



Per studiare le problematiche legate alla programmazione concorrente e i meccanismi di sincronizzazione ci concentreremo su uno scenario multithreading assumendo un singolo processo e una singola CPU

Scenario che consideriamo per studiare la concorrenza (multithreading)



Come al solito, ragioneremo sui modelli e sui costrutti linguistici, astraendo dai meccanismi che si attivano a livello hardware e di sistema operativo

Un modello della programmazione concorrente

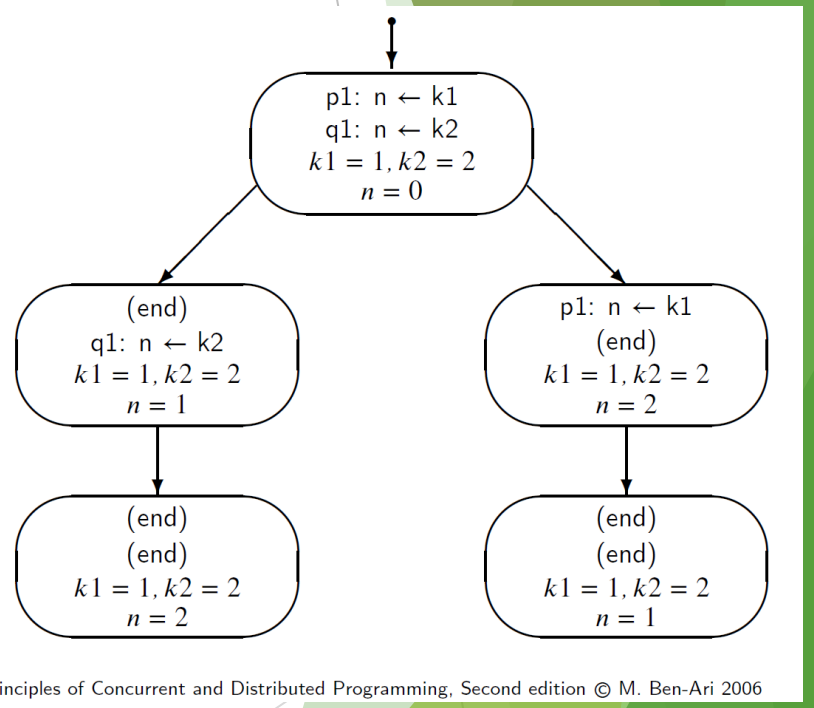
Scopo del modello

L'insieme dei comportamenti possibili di un programma concorrente può essere descritto come un sistema di transizioni

- Interleaving e scelte non deterministiche

Il sistema di transizioni è definito dalla semantica del linguaggio di programmazione

Usando un modello potremo analizzare le problematiche di sincronizzazione sul sistema di transizioni di un linguaggio semplice



M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

La base:
un linguaggio imperativo minimale

Sintassi

$$e ::= n \mid e + e \mid !\ell \mid \ell := e \mid skip \mid e; e$$

Tipi int, unit

In sostanza

- somme di interi, lettura (!) e scrittura (:=) di locazioni di memoria, skip e sequenze (;)
- assegnamenti come espressioni di tipo unit con side effect (come in Ocaml)

La base:
un linguaggio imperativo minimale

Regole di tipo

$$\Gamma \vdash n : \text{int} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \qquad \frac{\Gamma(\ell) = \text{intloc}}{\Gamma \vdash !\,\ell : \text{int}}$$

$$\frac{\Gamma(\ell) = \text{intloc} \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash \ell := e : \text{unit}} \qquad \frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash e_1 ; e_2 : \text{unit}}$$

$$\Gamma \vdash skip : \text{unit}$$

Semantica:

(op+) $\langle n_1 + n_2, s\rangle \to \langle n, s\rangle$ if $n = n_1 + n_2$

(comp1) $$\frac{\langle e_1,s\rangle \to \langle e_1',s'\rangle}{\langle e_1+e_2,s\rangle \to \langle e_1'+e_2,s'\rangle}$$

(comp2) $$\frac{\langle e_2,s\rangle \to \langle e_2',s'\rangle}{\langle n+e_2,s\rangle \to \langle n+e_2',s'\rangle}$$

(deref) $\langle !\,\ell, s\rangle \to \langle n, s\rangle$ if $\ell \in dom(s)$ and $s(l) = n$

(assign1) $\langle \ell := n\ , s\rangle \to \langle skip, s + \{\ell \mapsto n\}\rangle$ if $\ell \in dom(s)$

(assign2) $$\frac{\langle e,s\rangle \to \langle e',s'\rangle}{\langle \ell := e,s\rangle \to \langle \ell := e',s'\rangle}$$

(seq1) $\langle skip\ ;\ e_2\ , s\rangle \to \langle e_2, s\rangle$

(seq2) $$\frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle e_1; e_2\ , s\rangle \to \langle e_1'; e_2\ , s'\rangle}$$

Esempio

Eseguiamo il programma

$$\ell_1 := 3; \ell_2 := !\,\ell_1 + 1$$

nella memoria

$$\{\ell_1 \mapsto 0; \ell_2 \mapsto 0\}$$

Risultato

$$\langle \ell_1 := 3; \ell_2 := !\,\ell_1 + 1 , \{\ell_1 \mapsto 0; \ell_2 \mapsto 0\}\rangle \rightarrow$$
$$\langle skip; \ell_2 := !\,\ell_1 + 1 , \{\ell_1 \mapsto 3; \ell_2 \mapsto 0\}\rangle \rightarrow$$
$$\langle \ell_2 := !\,\ell_1 + 1 , \{\ell_1 \mapsto 3; \ell_2 \mapsto 0\}\rangle \rightarrow$$
$$\langle \ell_2 := 3 + 1 , \{\ell_1 \mapsto 3; \ell_2 \mapsto 0\}\rangle \rightarrow$$
$$\langle \ell_2 := 4, \{\ell_1 \mapsto 3; \ell_2 \mapsto 0\}\rangle \rightarrow$$
$$\langle skip, \{\ell_1 \mapsto 3; \ell_2 \mapsto 4\}\rangle$$

Estensione concorrente

Estensione concorrente: scelte di progettazione

- thread con memoria condivisa (shared memory)
- thread eseguiti in modo concorrente con interleaving
- thread privi di identità
- la terminazione di un thread non è osservabile dal resto del programma
- thread non possono essere forzati a terminare dall'esterno
- thread non restituiscono un risultato al termine

Estensione concorrente

Modelleremo i thread come espressioni composte tramite un operatore di parallel composition: $e_1 | e_2$

- e_1 ed e_2 sono i due thread in esecuzione concorrente

Ambiguità terminologiche…

- l'operatore | si chiama tradizionalmente "parallel" composition, ma si interpreta come composizione concorrente…
- anche gli operandi di | sono tradizionalmente chiamati "processi" anche se in questo caso il loro comportamento sarà più simile a quello dei thread

Estensione concorrente

Sintassi estesa

$$e ::= n \mid e + e \mid !\ell \mid \ell := e \mid skip \mid e; e \mid e|e$$

Tipi estesi int, unit, proc

Regole di tipo aggiunte

$$\frac{\Gamma \vdash e\text{:unit}}{\Gamma \vdash e\text{:proc}}$$

$$\frac{\Gamma \vdash e_1\text{:proc} \qquad \Gamma \vdash e_2\text{:proc}}{\Gamma \vdash e_1 \mid e_2\text{:proc}}$$

Estensione concorrente

Regole semantiche aggiunte

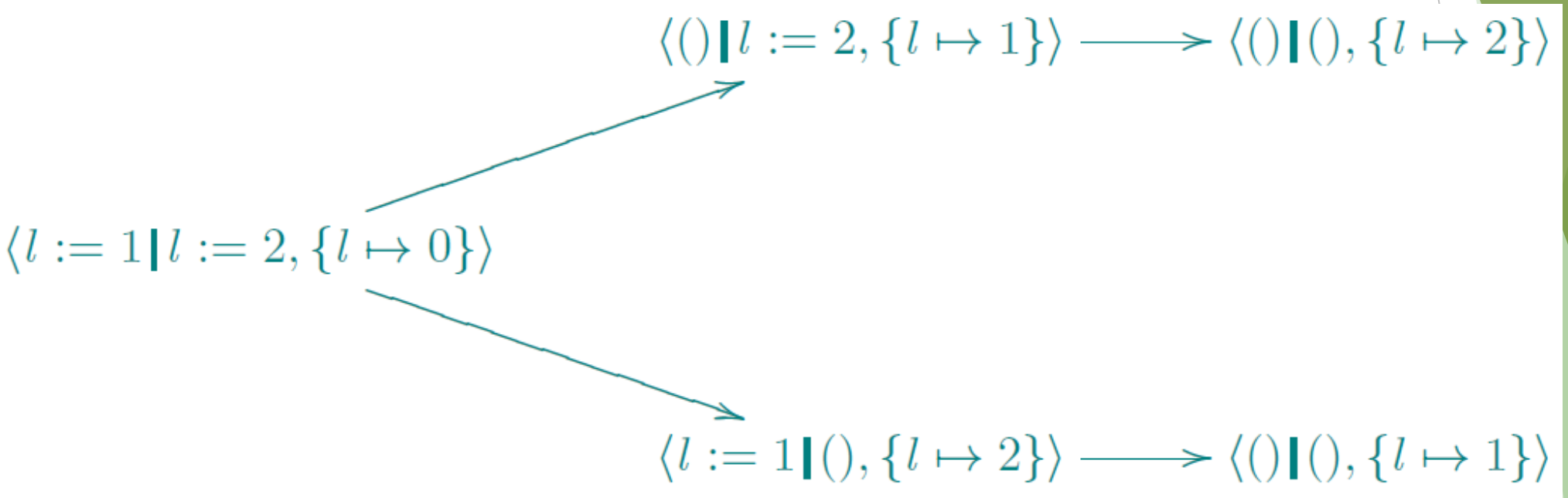
(parallel1) $$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1 \,|\, e_2, s\rangle \longrightarrow \langle e_1' \,|\, e_2, s'\rangle}$$

(parallel2) $$\frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle e_1 \,|\, e_2, s\rangle \longrightarrow \langle e_1 \,|\, e_2', s'\rangle}$$

Queste regole descrivono in modo semplice il meccanismo di interleaving

- due (o più) thread concorrenti possono avanzare uno per volta
- la scelta tra l'applicazione di (parallel1) e (parallel2) è non deterministica
- la memoria è condivisa

Esempio concorrente



Nota: () è l'abbreviazione di skip

Operazioni atomiche

- Per come abbiamo definito la semantica, le operazioni atomiche del linguaggio (quelle che danno origine ad una transizione) sono
  - somma (+)
  - dereferenziazione (!)
  - assegnamento (:=)
  - skip

Operazioni atomiche

- Per come abbiamo definito la semantica, le operazioni atomiche del linguaggio (quelle che danno origine ad una transizione) sono
  - somma (+)
  - dereferenziazione (!)
  - assegnamento (:=)
  - skip

In particolare, lettura e scrittura in memoria sono operazioni atomiche separate

(come a livello macchina… vedere discorsi precedenti)

La semantica cattura bene i comportamenti a basso livello

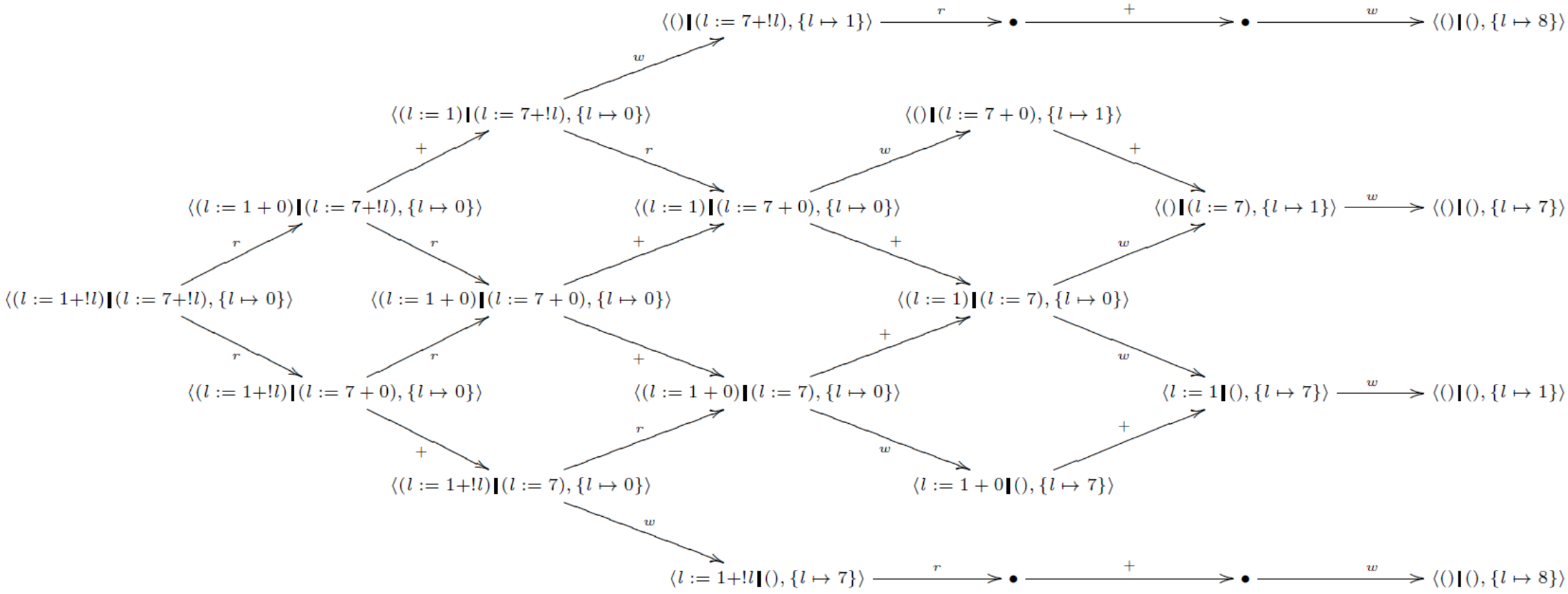
Esempio concorrente "problematico"

Eseguiamo $(\ell := 1 + !\,\ell) | (\ell := 7 + !\,\ell)$ nella memoria $\{\ell \mapsto 0\}$

- Sono due thread che condividono la locazione ℓ
- Ognuno cerca di incrementare il valore in locazione ℓ
- Qual è il comportamento di questo programma?

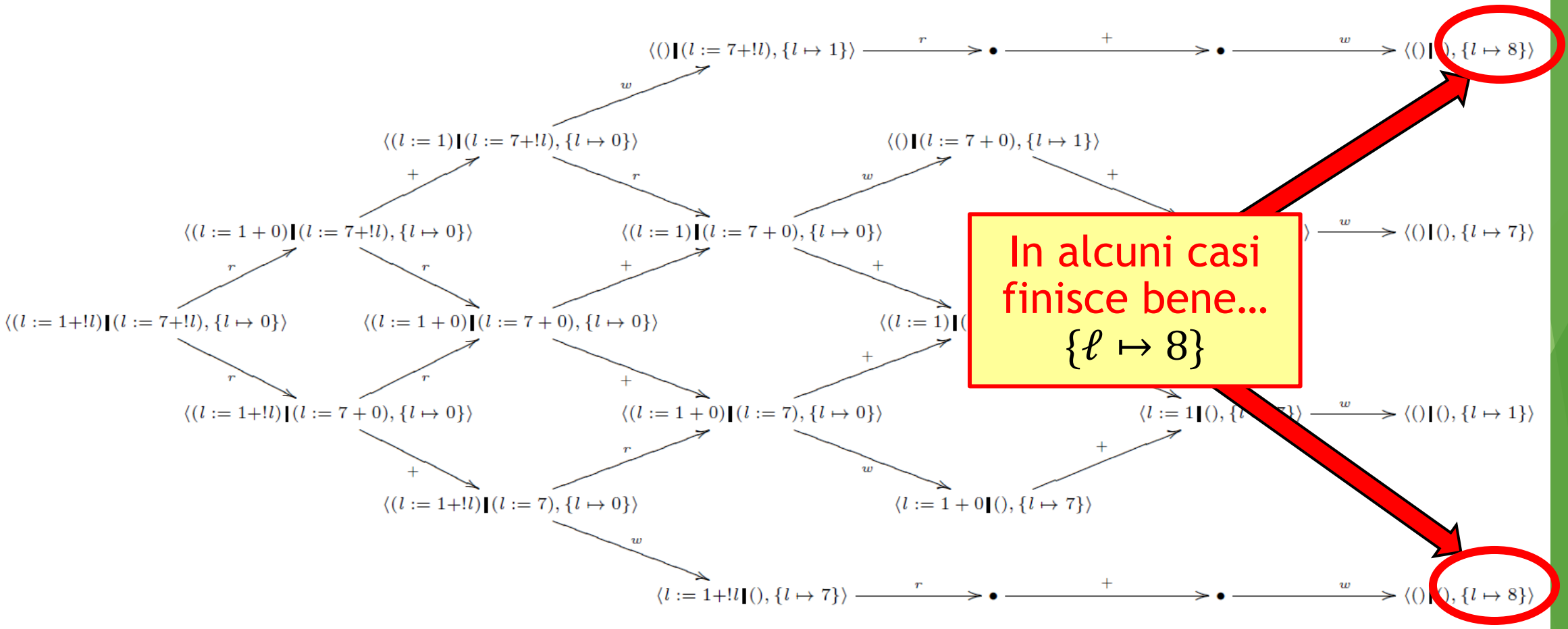
Esempio concorrente "problematico"

La semantica ci fornisce il sistema di transizioni che descrive tutti i comportamenti possibili!



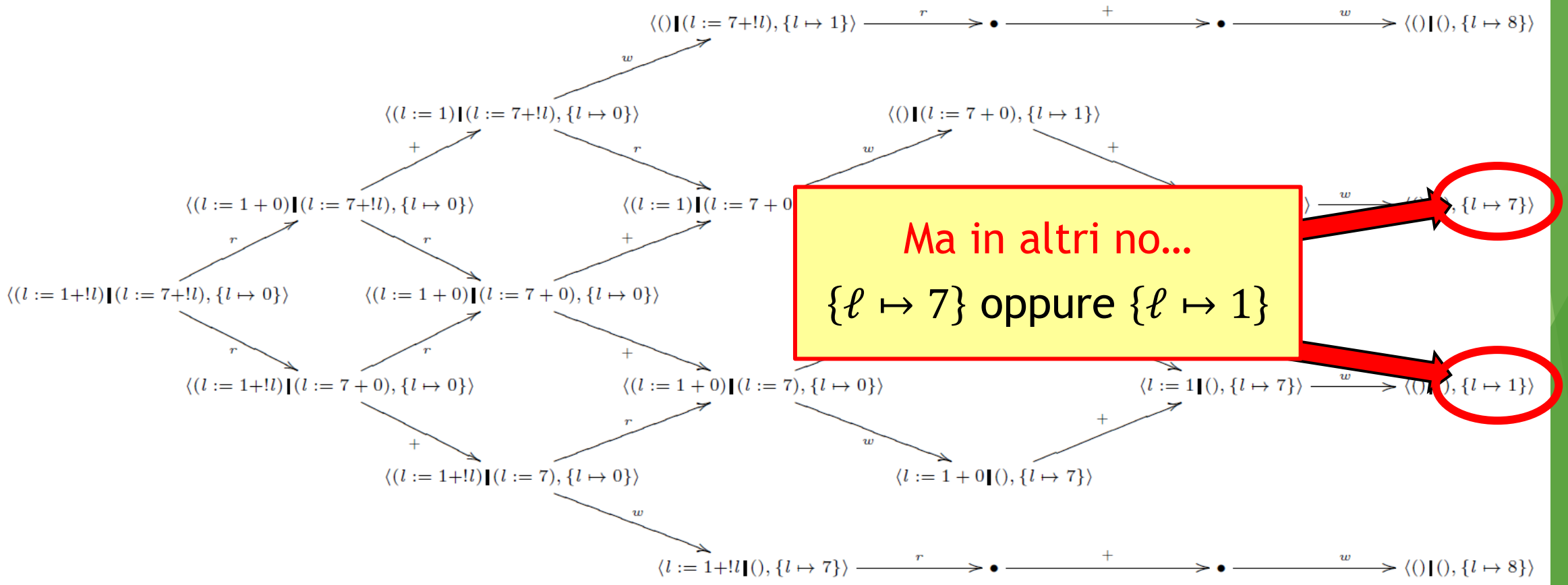
Esempio concorrente "problematico"

La semantica ci fornisce il sistema di transizioni che descrive tutti i comportamenti possibili!



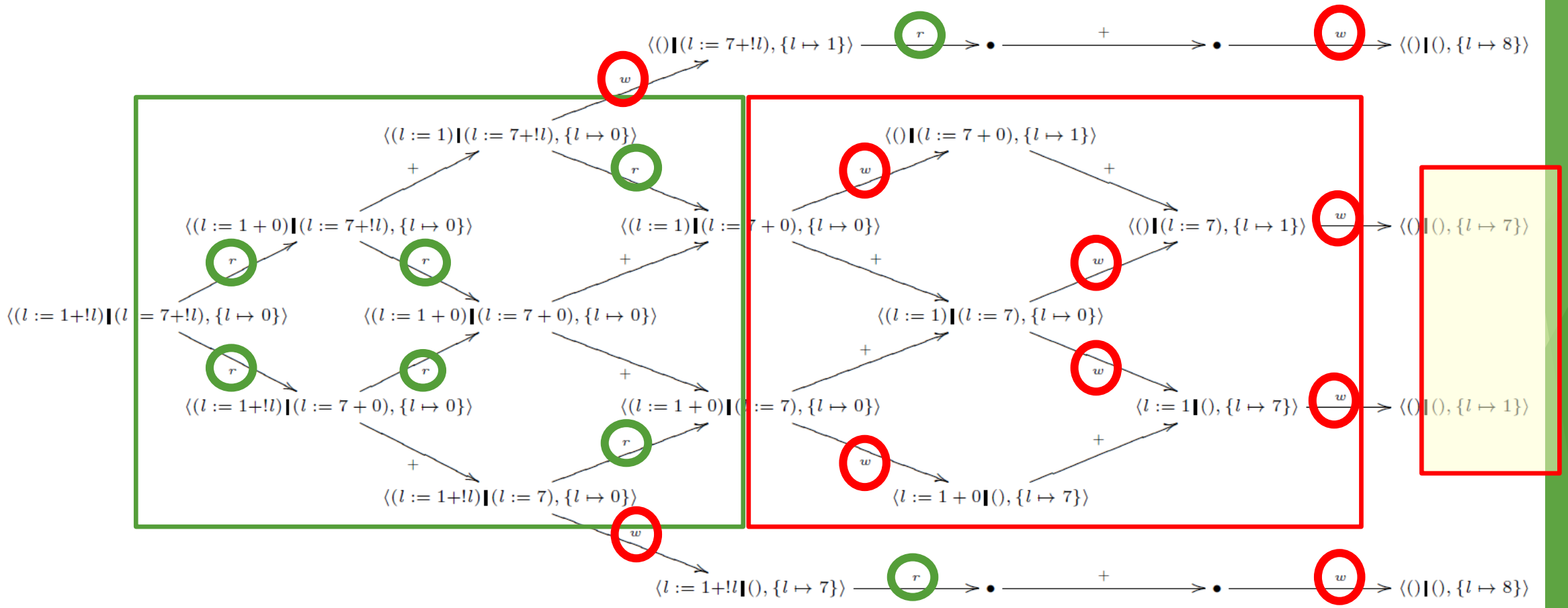
Esempio concorrente "problematico"

La semantica ci fornisce il sistema di transizioni che descrive tutti i comportamenti possibili!



Tutti (e soli) i cammini che portano a risultati sbagliati fanno prima tutte le letture (r) e poi tutte le scritture (w)!

Il sistema di transizioni mostra bene questa proprietà!



Osservazioni

Il sistema di transizioni consente di ragionare bene sulle proprietà comportamentali dei programmi concorrenti

Ma

- C'è un'esplosione combinatoriale dello spazio degli stati
- ossia, a causa dell'interleaving, il numero degli stati raggiungibili (dimensione del grafo) può crescere esponenzialmente rispetto alla dimensione del programma

L'analisi delle proprietà del comportamento dei programmi concorrenti richiede di applicare opportuni metodi di analisi statica

Meccanismi di sincronizzazione

Ma quindi come si fa a far funzionare correttamente il programma $(\ell := 1 + !\,\ell)|(\ell := 7 + !\,\ell)$???

- Bisogna introdurre nel linguaggio un meccanismo di sincronizzazione, per evitare che un thread legga il valore mentre l'altro lo sta per modificare
- Rimanendo nel modello, introduciamo un meccanismo di mutua esclusione basata su "lock", che astrae i vari meccanismi presenti nei linguaggi di programmazione reali

Un modello della programmazione concorrente (con i lock)

Primitive di Mutua Esclusione (mutex)

Nei linguaggi di programmazione esistono diverse primitive di sincronizzazione

Come esempi, abbiamo già visto

- sleep e wakeup
- send e receive
- …

Tutte quante prevedono meccanismi per bloccare e sbloccare processi/thread

Primitive di Mutua Esclusione (mutex)

Nel caso di thread concorrenti con shared memory, la primitiva di sincronizzazione di riferimento si basa sull'uso di meccanismi di locking

- Servono per assicurare mutua esclusione per l'accesso alle locazioni di memoria condivise
  - un thread può assicurarsi per un po' di passi di essere l'unico ad accedere ad una certa locazione di memoria condivisa
- Prevedono le primitive lock e unlock

Primitive di Mutua Esclusione (mutex)

Come funziona

Ad ogni area di memoria da condividere e da accedere in mutua esclusione è associato un mutex m

- una entità astratta (un token, un testimone,…)

- Prima di accedere all'area di memoria un thread T1 deve acquisire il mutex m eseguendo lock m
  - Se nessun altro thread T2 sta accedendo a quell'area di memoria, T1 procede
  - In caso contrario, T1 si blocca in attesa che T2 (che aveva già acquisito il mutex m) lo rilasci eseguendo unlock m
- Dopo aver acceduto alla memoria, il thread T1 rilascia il mutex m eseguendo unlock m

Estendiamo il linguaggio concorrente con le primitive di locking

Sintassi estesa

$$e ::= n \mid e + e \mid !\ell \mid \ell := e \mid skip \mid e; e \mid e|e \mid \textbf{lock}\, m \mid \textbf{unlock}\, m$$

dove $m \in \mathbb{M}$ (insieme, anche infinito, dei mutex)

Regole di tipo aggiunte

$$\Gamma \vdash \textbf{lock}\ \ m\text{:unit} \qquad \Gamma \vdash \textbf{unlock}\ \ m\text{:unit}$$

Configurazioni della semantica

- invece di $\langle e, s \rangle$ abbiamo $\langle e, s, M \rangle$ con $M: \mathbb{M} \mapsto \{true, false\}$
- $M(m)$ rappresenta lo stato del mutex m (è $true$ se già acquisito)

Estendiamo il linguaggio concorrente con le primitive di locking

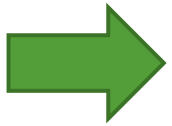
Regole semantiche aggiunte

(lock) $\langle \textbf{lock}\ \ m, s, M\rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{true}\}\rangle$ if $\neg M(m)$

(unlock) $\langle \textbf{unlock}\ \ m, s, M\rangle \longrightarrow \langle (), s, M + \{m \mapsto \textbf{false}\}\rangle$

Inoltre tutte le regole semantiche precedentemente definite vanno modificate aggiungendo M ovunque… ad esempio

(seq2) $$\frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle e_1;e_2\ , s\rangle \to \langle e_1';e_2\ , s'\rangle}$$



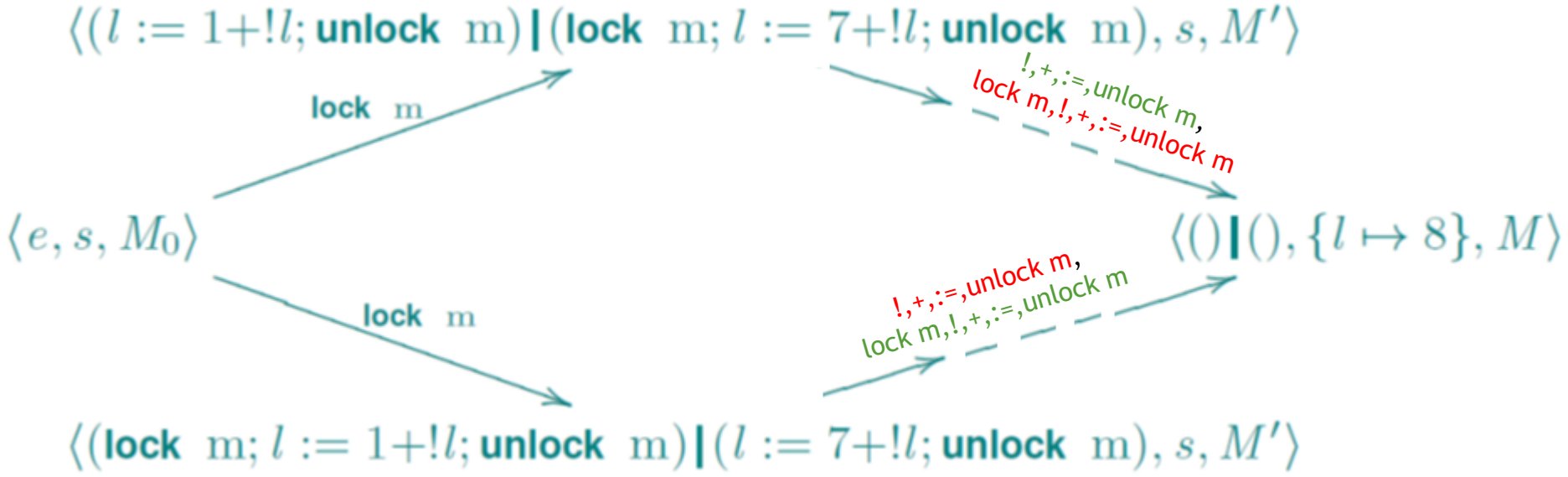
$$\frac{\langle e_1, s, M\rangle \to \langle e_1', s', M'\rangle}{\langle e_1;e_2, s, M\rangle \to \langle e_1';e_2\ , s', M'\rangle}$$

Usare i mutex

Riprendiamo l'esempio: $(\ell := 1 + !\,\ell)|(\ell := 7 + !\,\ell)$ e ora usiamo un mutex

$$e = (\mathbf{lock}\, m; \ell := 1 + !\,\ell; \mathbf{unlock}\, m)|(\mathbf{lock}\, m; \ell := 7 + !\,\ell; \mathbf{unlock}\, m)$$

Il comportamento di e nella memoria $s = \{\ell \mapsto 0\}$ e con stato iniziale dei mutex $M_0 = \{\forall m.\,(m \mapsto false)\}$ è



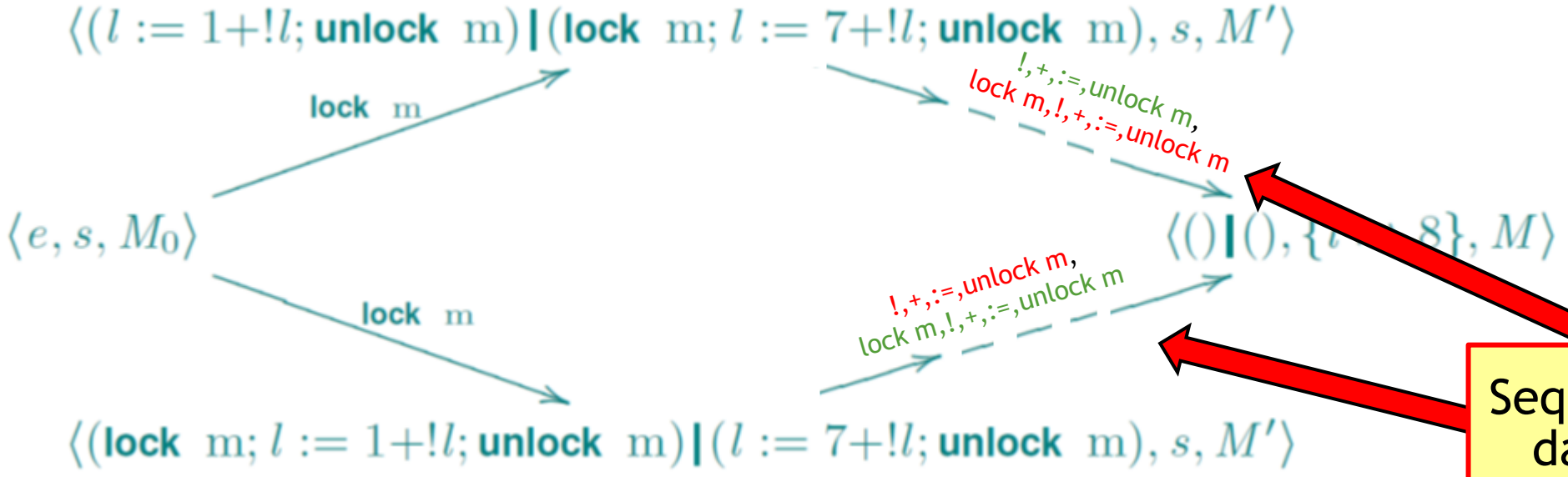
dove $M' = M_0 + \{\mathrm{m} \mapsto \mathbf{true}\}$

Usare i mutex

Riprendiamo l'esempio: $(\ell := 1 + !\,\ell)|(\ell := 7 + !\,\ell)$ e ora usiamo un mutex

$$e = (\mathbf{lock}\, m; \ell := 1 + !\,\ell; \mathbf{unlock}\, m)|(\mathbf{lock}\, m; \ell := 7 + !\,\ell; \mathbf{unlock}\, m)$$

Il comportamento di e nella memoria $s = \{\ell \mapsto 0\}$ e con stato iniziale dei mutex $M_0 = \{\forall m.(m \mapsto false)\}$ è



dove $M' = M_0 + \{\mathrm{m} \mapsto \mathbf{true}\}$

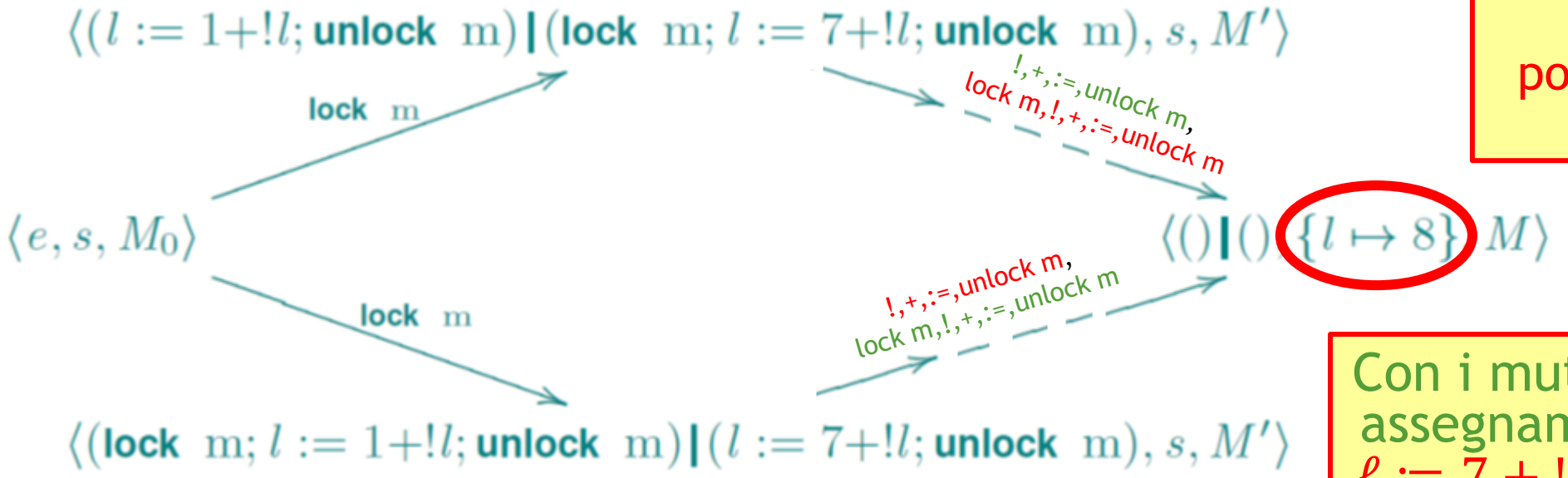
Sequenze di passi fatti dal primo thread e secondo thread

Usare i mutex

Riprendiamo l'esempio: $(\ell := 1 + !\,\ell)|(\ell := 7 + !\,\ell)$ e ora usiamo un mutex

$$e = (\mathbf{lock}\, m; \ell := 1 + !\,\ell; \mathbf{unlock}\, m)|(\mathbf{lock}\, m; \ell := 7 + !\,\ell; \mathbf{unlock}\, m)$$

Il comportamento di e nella memoria $s = \{\ell \mapsto 0\}$ e con stato iniziale dei mutex $M_0 = \{\forall m.(m \mapsto false)\}$ è



dove $M' = M_0 + \{\mathrm{m} \mapsto \mathbf{true}\}$

Sono sparite le esecuzioni che portavano a risultati "sbagliati"

Con i mutex ha eseguito gli assegnamenti $\ell := 1 + !\,\ell$ e $\ell := 7 + !\,\ell$ come se fossero atomici

Usare i mutex

- La chiave per il funzionamento dei mutex è nel fatto che la primitiva lock può bloccare il thread
- Un uso coerente dei mutex in tutti i thread previene la possibilità di interleaving non desiderati

Principi per un uso coerente dei mutex

- Ad ogni locazione condivisa a cui accedere è associato un mutex (un mutex può essere associato a più locazioni)
- Prima di iniziare ad utilizzare la locazione, il thread fa lock sul mutex ad essa associato
- Dopo aver terminato di utilizzare la locazione, il thread fa unlock sul mutex ad essa associato

"Coarse-grained" and "fine-grained" locking

L'associazione dei mutex alle locazioni può essere fatta in modi diversi

Esempi ESTREMI:

- Unico mutex per tutte le locazioni (coarse-grained)

  $\ell_1, \ell_2, \ldots \mapsto m$

- Un mutex diverso per ogni locazione (fine-grained)

  $\ell_1 \mapsto m_1, \ell_2 \mapsto m_2 , \ldots$

"Coarse-grained" and "fine-grained" locking

Vediamo un esempio: thread che accede a due locazioni

- coarse-grained

$$e_{cg} = (\mathbf{lock}\, m; \ell_1 := 1 + !\,\ell_2; \mathbf{unlock}\, m)$$

- fine-grained

$$e_{fg} = (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_1 := 1 + !\,\ell_2; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$

La strategia coarse-grained

- richiede meno lavoro al singolo thread (una singola operazione di lock)
- ma riduce la concorrenza ($\mathbf{lock}\, m$ blocca tutti gli altri thread, anche quelli che accedono a locazioni di memoria diverse)

"Coarse-grained" and "fine-grained" locking

Un altro esempio (idealmente, il programma: x++;y++|x++;z++)

- coarse-grained

$$e'_{cg} = (\mathbf{lock}\, m; \ell_1 := 1 + !\,\ell_1; \ell_2 := 1 + !\,\ell_2; \mathbf{unlock}\, m)$$
$$|\,(\mathbf{lock}\, m; \ell_1 := 1 + !\,\ell_1\,; \ell_3 := 1 + !\,\ell_3; \mathbf{unlock}\, m)$$

- fine-grained

$$e'_{fg} = (\mathbf{lock}\, m_1; \ell_1 := 1 + !\,\ell_1; \mathbf{unlock}\, m_1; \mathbf{lock}\, m_2; \ell_2 := 1 + !\,\ell_2;\ \mathbf{unlock}\, m_2)$$
$$|\,(\mathbf{lock}\, m_1; \ell_1 := 1 + !\,\ell_1\,; \mathbf{unlock}\, m_1; \mathbf{lock}\, m_3; \ell_3 := 1 + !\,\ell_3; \mathbf{unlock}\, m_3)$$

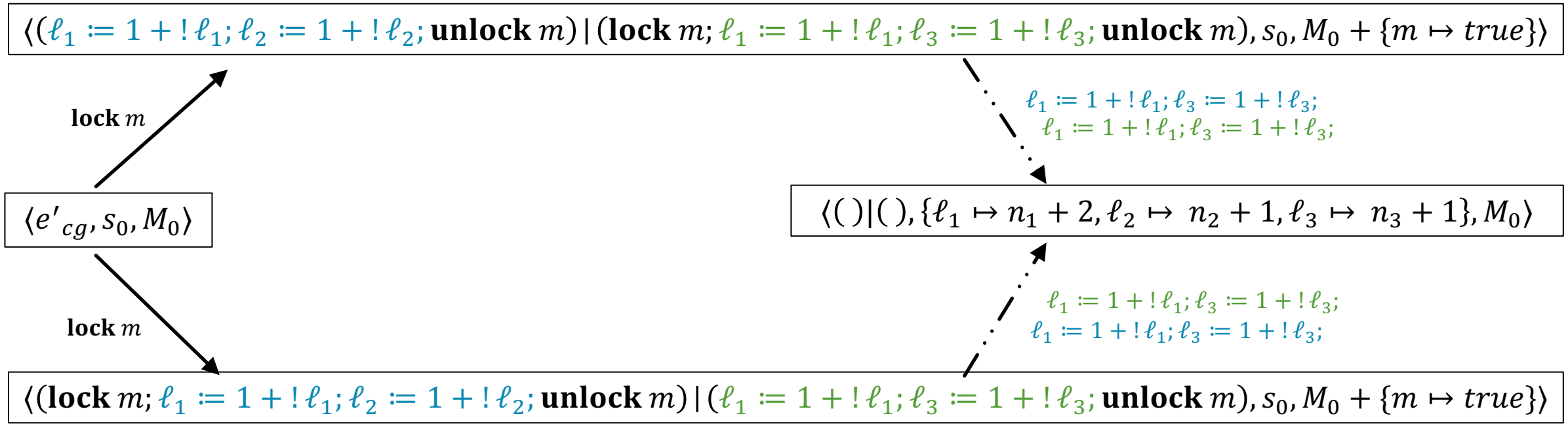
Esecuzione coarse-grained

$$e'_{cg} = (\mathbf{lock}\, m; \ell_1 := 1 + !\,\ell_1; \ell_2 := 1 + !\,\ell_2; \mathbf{unlock}\, m) \mid (\mathbf{lock}\, m; \ell_1 := 1 + !\,\ell_1; \ell_3 := 1 + !\,\ell_3; \mathbf{unlock}\, m)$$

$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2, \ell_3 \mapsto n_3\}$$

$$M_0 = \{\forall m.\, (m \mapsto false)\}$$

Sistema di transizioni (possibili esecuzioni)



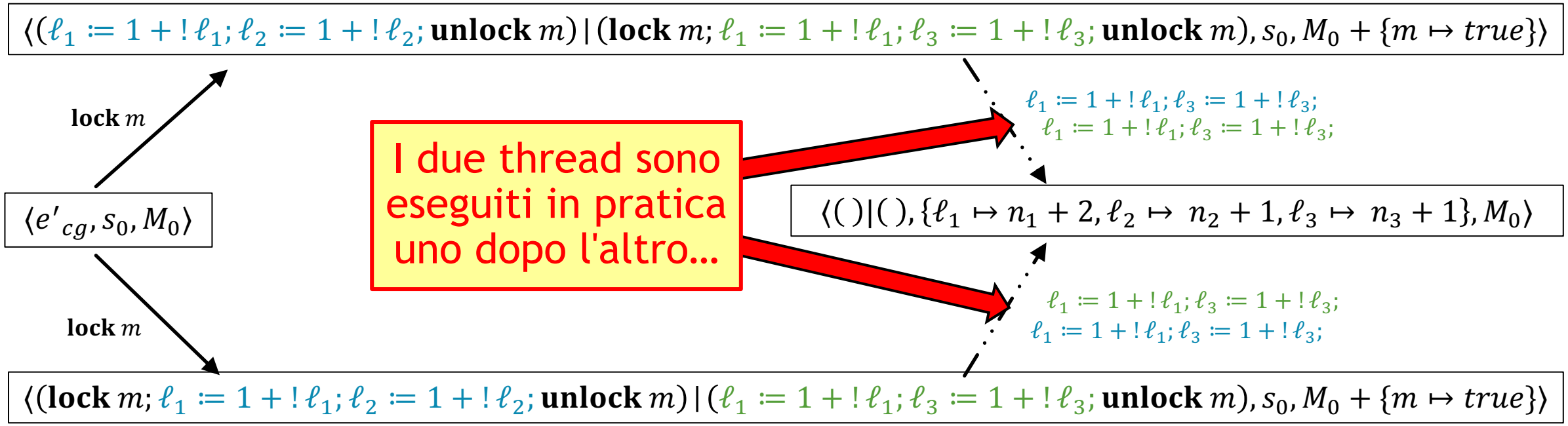
Esecuzione coarse-grained

$$e'_{cg} = (\mathbf{lock}\, m; \ell_1 := 1 + !\,\ell_1; \ell_2 := 1 + !\,\ell_2; \mathbf{unlock}\, m) \mid (\mathbf{lock}\, m; \ell_1 := 1 + !\,\ell_1; \ell_3 := 1 + !\,\ell_3; \mathbf{unlock}\, m)$$

$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2, \ell_3 \mapsto n_3\}$$

$$M_0 = \{\forall m.\, (m \mapsto false)\}$$

Sistema di transizioni (possibili esecuzioni)



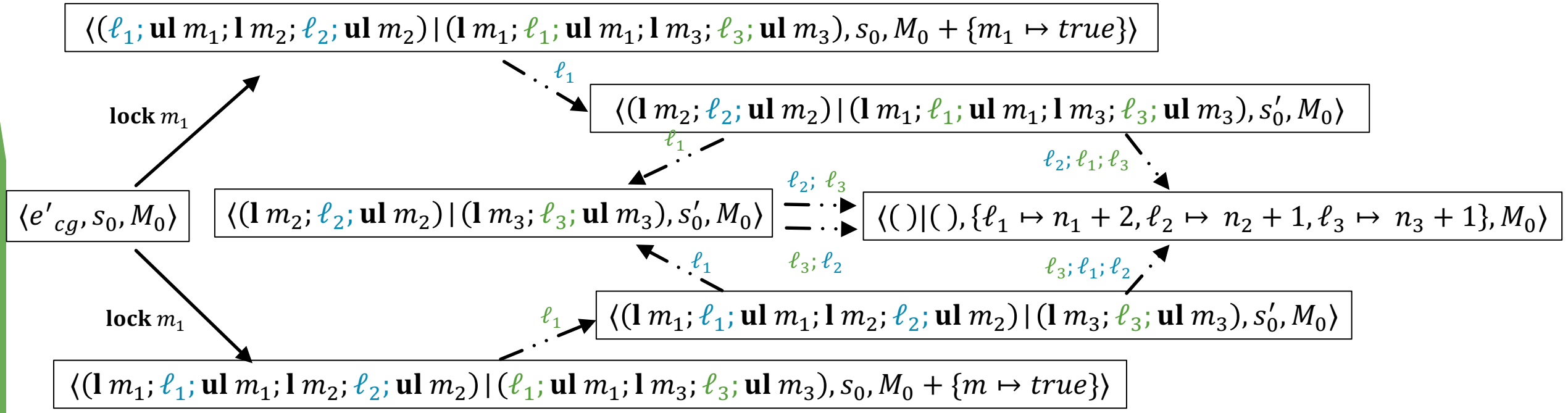
Esecuzione fine-grained

$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2, \ell_3 \mapsto n_3\}$$

$$M_0 = \{\forall m.\,(m \mapsto false)\}$$

$$e'_{fg} = (\mathbf{lock}\, m_1; \ell_1 := 1 + !\,\ell_1; \mathbf{unlock}\, m_1; \mathbf{lock}\, m_2; \ell_2 := 1 + !\,\ell_2;\ \mathbf{unlock}\, m_2)$$
$$\mid (\mathbf{lock}\, m_1; \ell_1 := 1 + !\,\ell_1\,; \mathbf{unlock}\, m_1; \mathbf{lock}\, m_3; \ell_3 := 1 + !\,\ell_3; \mathbf{unlock}\, m_3)$$

Sistema di transizioni (possibili esecuzioni)



Esecuzione fine-grained

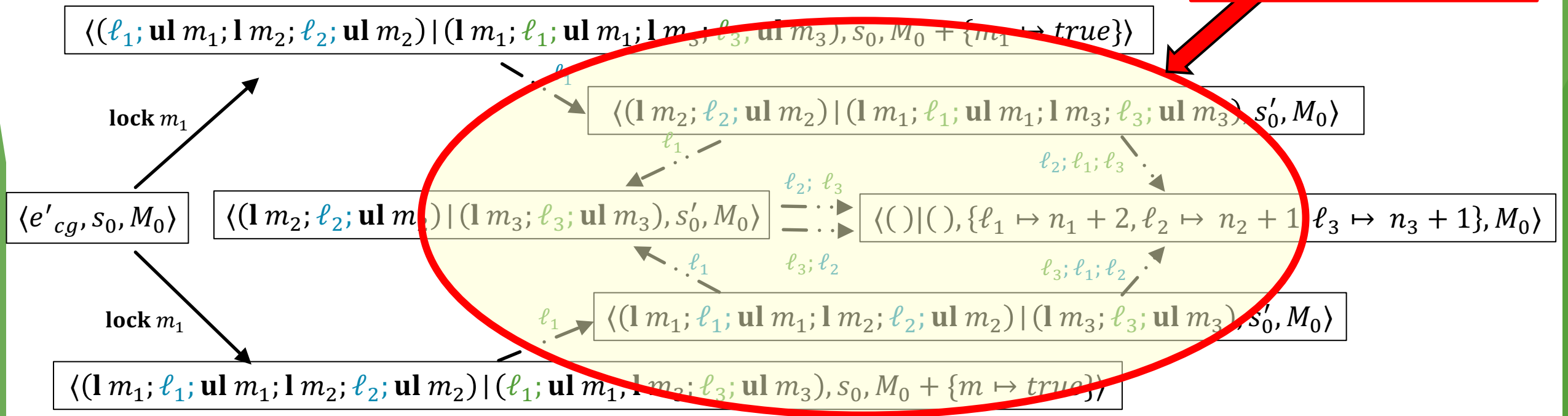
$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2, \ell_3 \mapsto n_3\}$$

$$M_0 = \{\forall m.\,(m \mapsto false)\}$$

$$e'_{fg} = (\mathbf{lock}\ m_1; \ell_1 := 1 + !\,\ell_1; \mathbf{unlock}\ m_1; \mathbf{lock}\ m_2; \ell_2 := 1 + !\,\ell_2;\ \mathbf{unlock}\ m_2)$$
$$|\,(\mathbf{lock}\ m_1; \ell_1 := 1 + !\,\ell_1\ ; \mathbf{unlock}\ m_1; \mathbf{lock}\ m_3; \ell_3 := 1 + !\,\ell_3; \mathbf{unlock}\ m$$

Maggiore libertà per l'esecuzione concorrente (o parallela)

Sistema di transizioni (possibili esecuzioni)



Uso pericoloso dei lock…

Le strategie fine-grained sono quindi spesso preferibili, ma hanno un altro problema…

$$e = (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$
$$\mid\ (\mathbf{lock}\, m_2; \mathbf{lock}\, m_1; \ell_2 := !\,\ell_1; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$

Che comportamento ha questo programma?

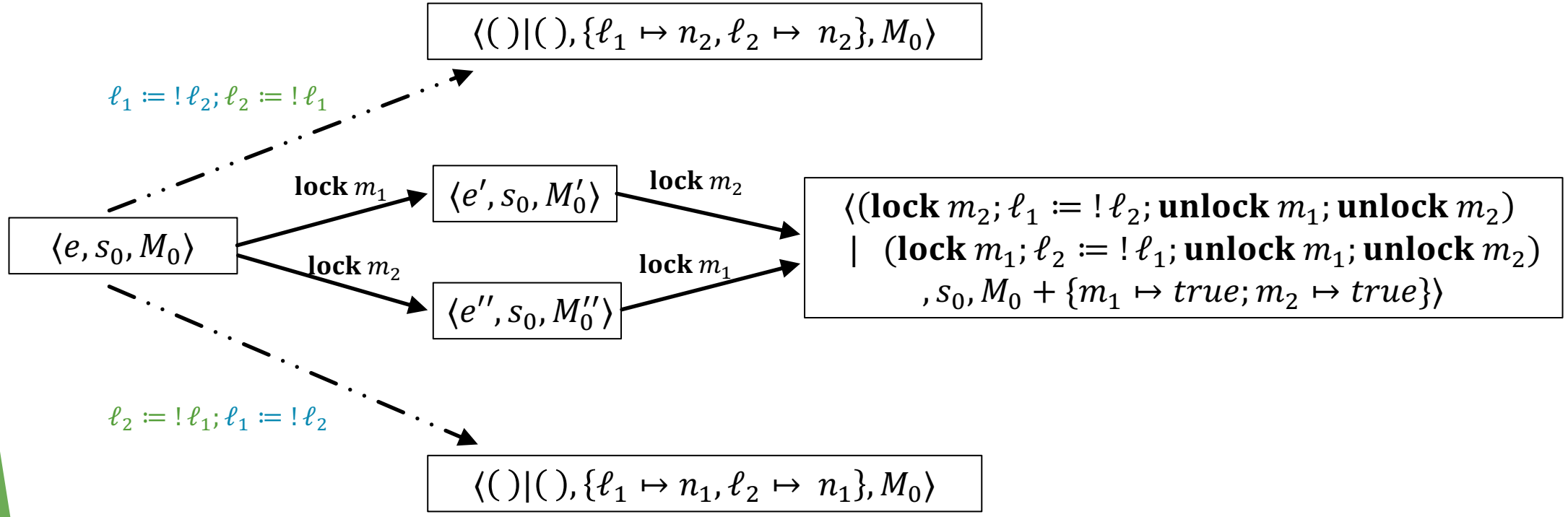
Uso pericoloso dei lock…

$$e = (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$
$$\mid\ (\mathbf{lock}\, m_2; \mathbf{lock}\, m_1; \ell_2 := !\,\ell_1; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$

$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2\}$$

$$M_0 = \{\forall m.\, (m \mapsto false)\}$$

Sistema di transizioni (possibili esecuzioni)



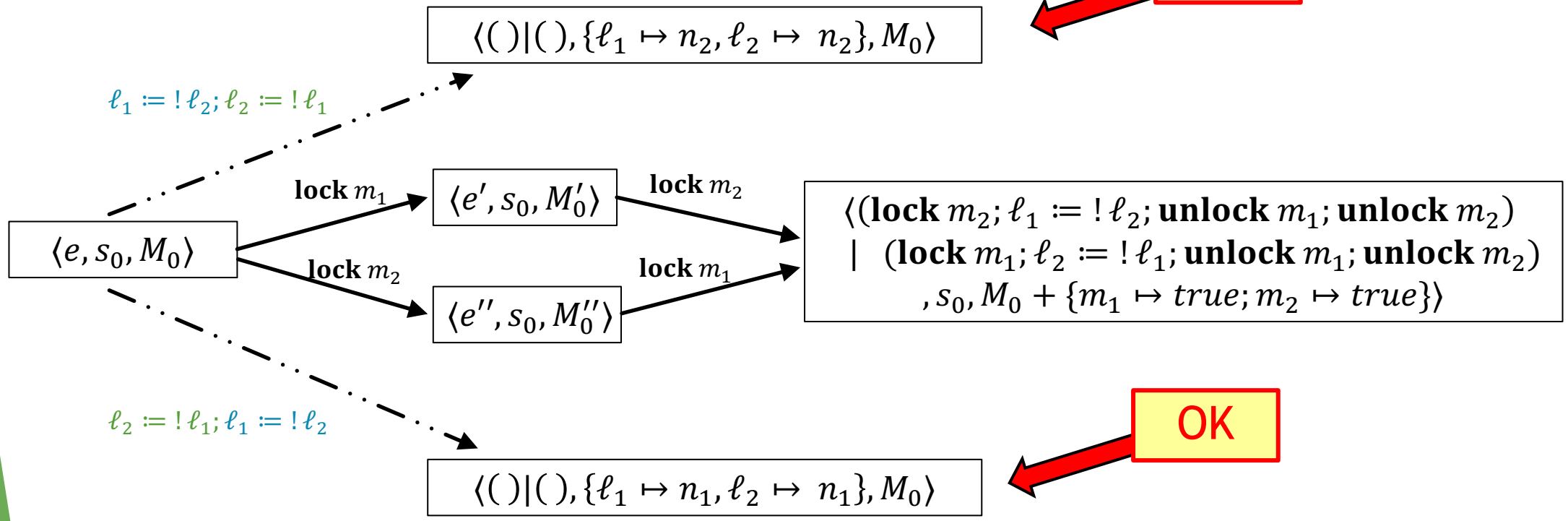
Uso pericoloso dei lock…

$$e = (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$
$$\mid\ (\mathbf{lock}\, m_2; \mathbf{lock}\, m_1; \ell_2 := !\,\ell_1; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$

$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2\}$$

$$M_0 = \{\forall m.\, (m \mapsto false)\}$$

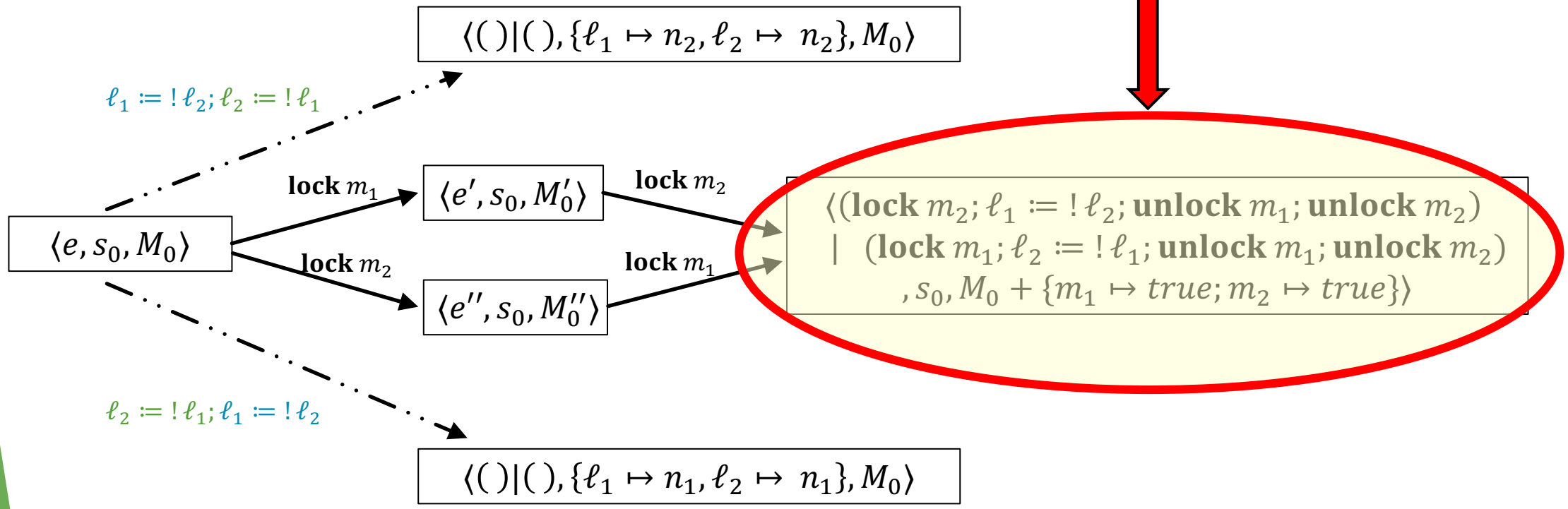
Sistema di transizioni (possibili esecuzioni)



Uso pericoloso dei lock…

$$e = (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$
$$\mid\ (\mathbf{lock}\, m_2; \mathbf{lock}\, m_1; \ell_2 := !\,\ell_1; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$

Sistema di transizioni (possibili esecuzioni)



Il programma si BLOCCA

Ogni thread ha acquisito un mutex, ma è bloccato nell'attesa che si liberi l'altro

Deadlock

La situazione in cui il programma si blocca (definitivamente) a causa dell'uso improprio dei lock prende il nome di

DEADLOCK

- E' dato da situazioni di attesa circolare, come quella vista nell'esempio precedente (A aspetta B intanto che B aspetta A)
- E' uno dei motivi per cui a volte le applicazioni in esecuzione sul computer si piantano e devono essere "killate"

Come risolvere i problemi di deadlock

Tre strategie

- Deadlock prevention: si stabiliscono regole controllabili staticamente (dal compilatore) sull'uso dei lock che garantiscano che i deadlock non si possano verificare
- Deadlock avoidance: l'esecuzione del programma è monitorata dal supporto a runtime del linguaggio che si accorge di quando il programma sta per andare in deadlock e interviene cambiando l'ordine di esecuzione dei thread
- Deadlock recovery: il programma viene lasciato libero di andare in deadlock, ma se ciò accade il runtime del linguaggio se ne accorge e interviene per ripristinare uno stato senza deadlock

Esempio di deadlock prevention: 2-phase locking (2PL)

Una disciplina sull'uso dei lock che previene i deadlock è il 2-phase locking (2PL)

Assume che esista un ordinamento dei mutex: $m_1, m_2, \ldots$

Stabilisce che un thread che voglia acquisire e poi rilasciare un certo numero di mutex deve fare

- i rispettivi **lock** in ordine crescente
- i rispettivi **unlock** in ordine decrescente

Le sequenze di lock e unlock devono essere eseguite completamente

- non si può acquisire n mutex, rilasciarne solo una parte e acquisirne altri, neanche se si rispetta l'ordine

Esempio di deadlock prevention: 2-phase locking (2PL)

Assumiamo che esista una associazione 1:1 tra locazioni e mutex

- associamo ℓ_1 a m_1, ℓ_2 a m_2, ecc…

Assumiamo l'ordinamento $m_1, m_2, m_3, \ldots$

Come va modificato questo esempio che andava in deadlock?

$$e = (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$
$$\mid\ (\mathbf{lock}\, m_2; \mathbf{lock}\, m_1; \ell_2 := !\,\ell_1; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$

Esempio di deadlock prevention: 2-phase locking (2PL)

$$e = (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$
$$\mid\ (\mathbf{lock}\, m_2; \mathbf{lock}\, m_1; \ell_2 := !\,\ell_1; \mathbf{unlock}\, m_1; \mathbf{unlock}\, m_2)$$

NO

$$e' = (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\, m_2; \mathbf{unlock}\, m_1)$$
$$\mid\ (\mathbf{lock}\, m_1; \mathbf{lock}\, m_2; \ell_2 := !\,\ell_1; \mathbf{unlock}\, m_2; \mathbf{unlock}\, m_1)$$

SI

Ordine crescente

Ordine decrescente

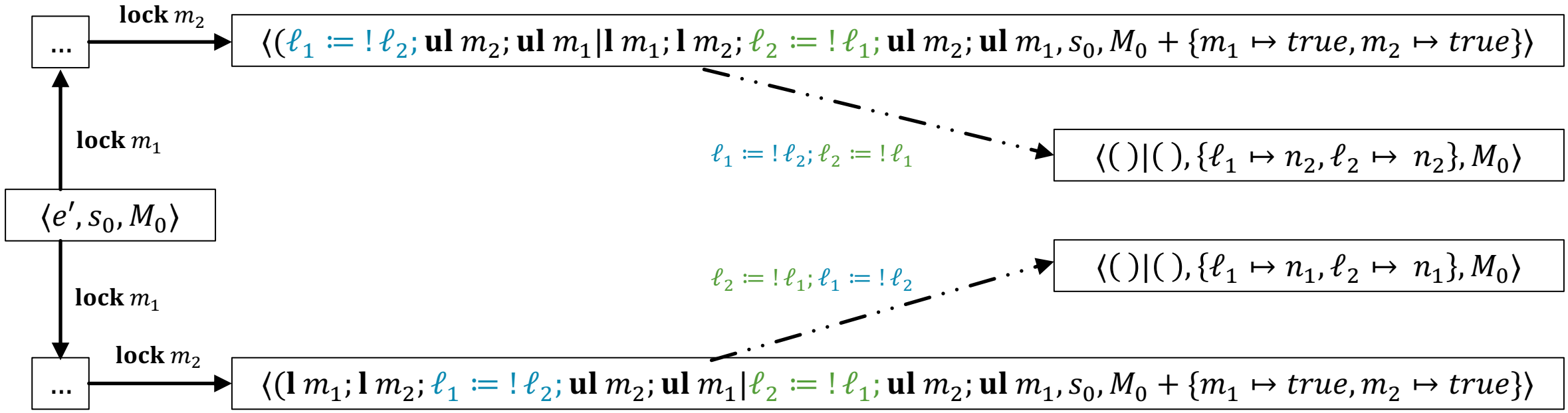
Esempio di deadlock prevention: 2-phase locking (2PL)

$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2, \ell_3 \mapsto n_3\}$$

$$M_0 = \{\forall m.\,(m \mapsto false)\}$$

$$e' = (\mathbf{lock}\ m_1; \mathbf{lock}\ m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\ m_2; \mathbf{unlock}\ m_1)$$
$$\mid\ (\mathbf{lock}\ m_1; \mathbf{lock}\ m_2; \ell_2 := !\,\ell_1; \mathbf{unlock}\ m_2; \mathbf{unlock}\ m_1)$$

Sistema di transizioni (possibili esecuzioni)



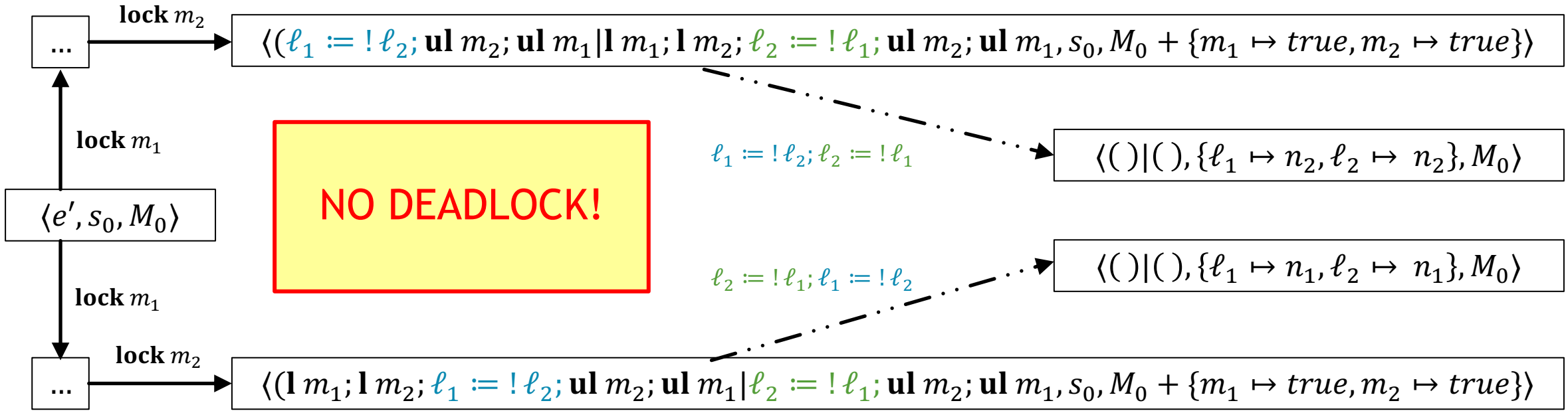
Esempio di deadlock prevention: 2-phase locking (2PL)

$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2, \ell_3 \mapsto n_3\}$$

$$M_0 = \{\forall m.\,(m \mapsto false)\}$$

$$e' = (\mathbf{lock}\ m_1; \mathbf{lock}\ m_2; \ell_1 := !\,\ell_2; \mathbf{unlock}\ m_2; \mathbf{unlock}\ m_1)$$
$$\mid\ (\mathbf{lock}\ m_1; \mathbf{lock}\ m_2; \ell_2 := !\,\ell_1; \mathbf{unlock}\ m_2; \mathbf{unlock}\ m_1)$$

Sistema di transizioni (possibili esecuzioni)



Riassumendo

Abbiamo definito un modello di programmazione concorrente che ci ha consentito di ragionare su alcuni concetti essenziali

- esecuzione non sequenziale
- mutua esclusione e sincronizzazione
- deadlock

Molti linguaggi di programmazione offrono primitive di concorrenza e sincronizzazione di alto livello, anche molto diverse tra loro

- aver visto i concetti su un modello semplice aiuterà nella comprensione dei costrutti di alto livello