Un modello della programmazione concorrente

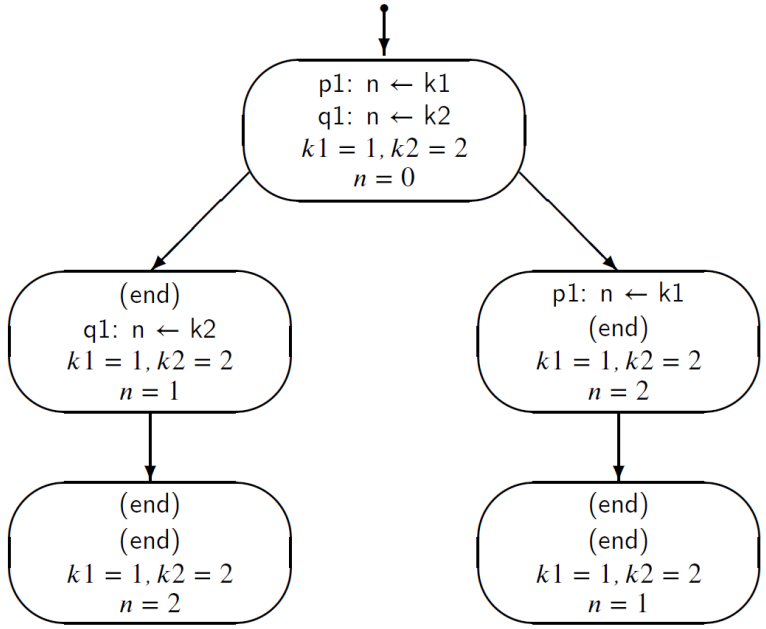
Scopo del modello

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 memora, 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:

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

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

Estensione concorrente

Regole semantiche aggiunte:

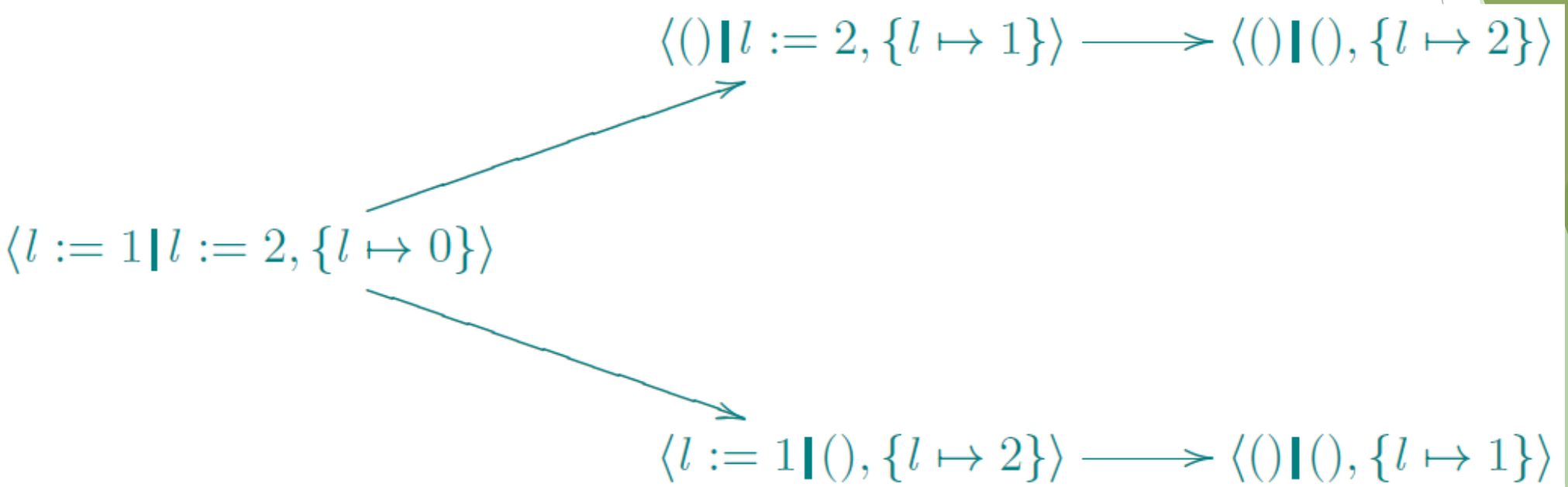
(parallel1)
$$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1 \,|\, e_2, s\rangle \longrightarrow \langle e_1' \,|\, e_2, s'\rangle}$$

(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

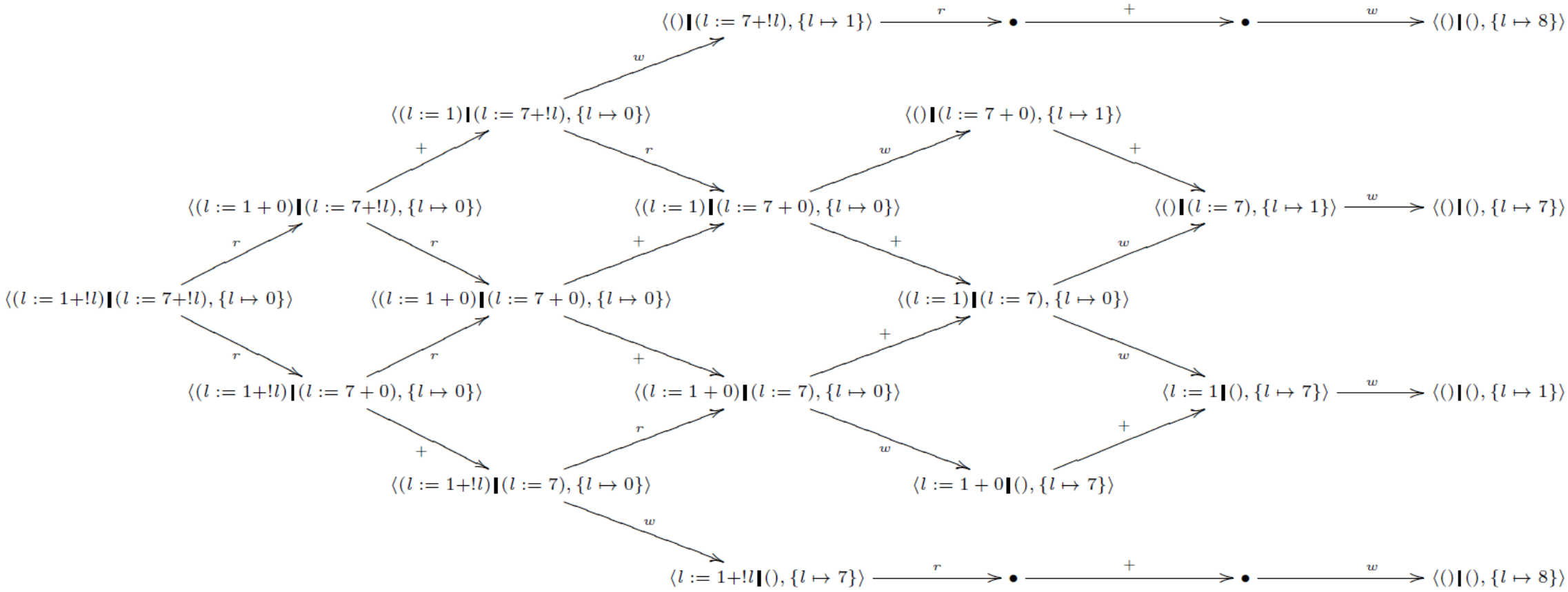
Esempio concorrente "problematico"

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?

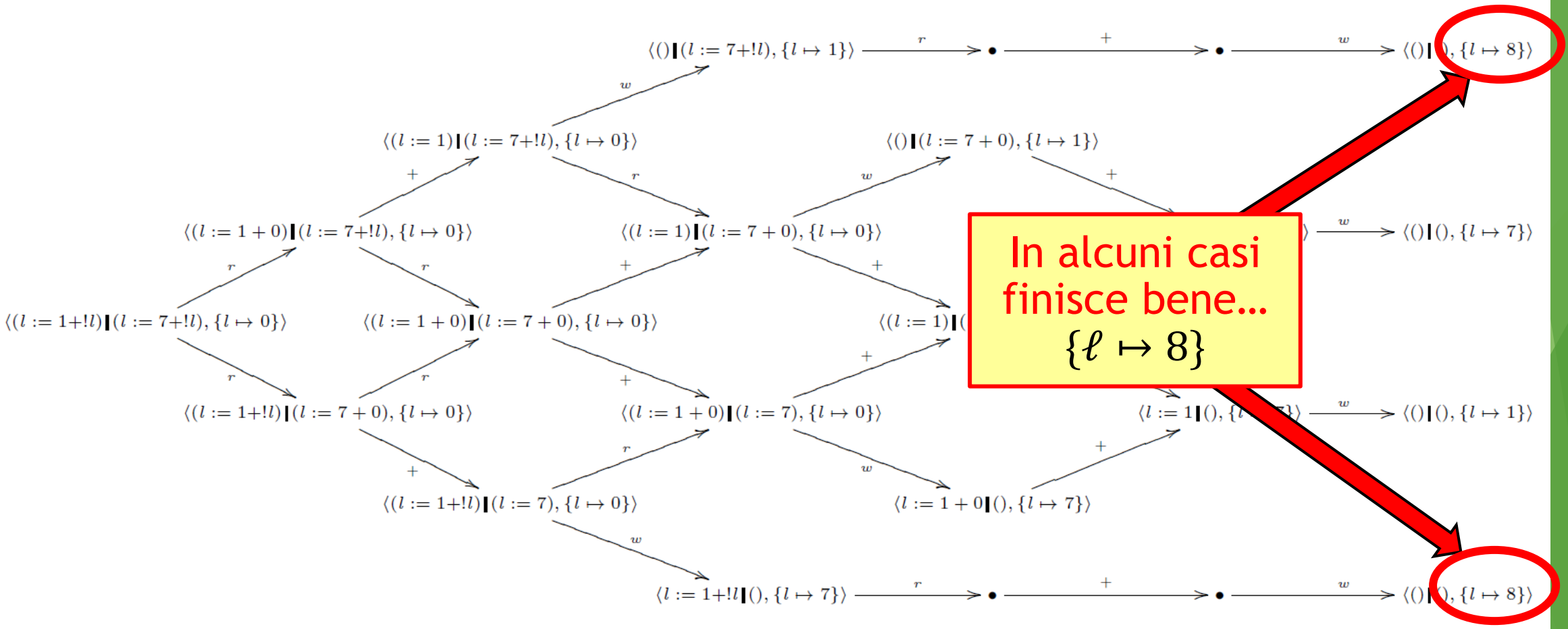
Esempio concorrente "problematico"

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



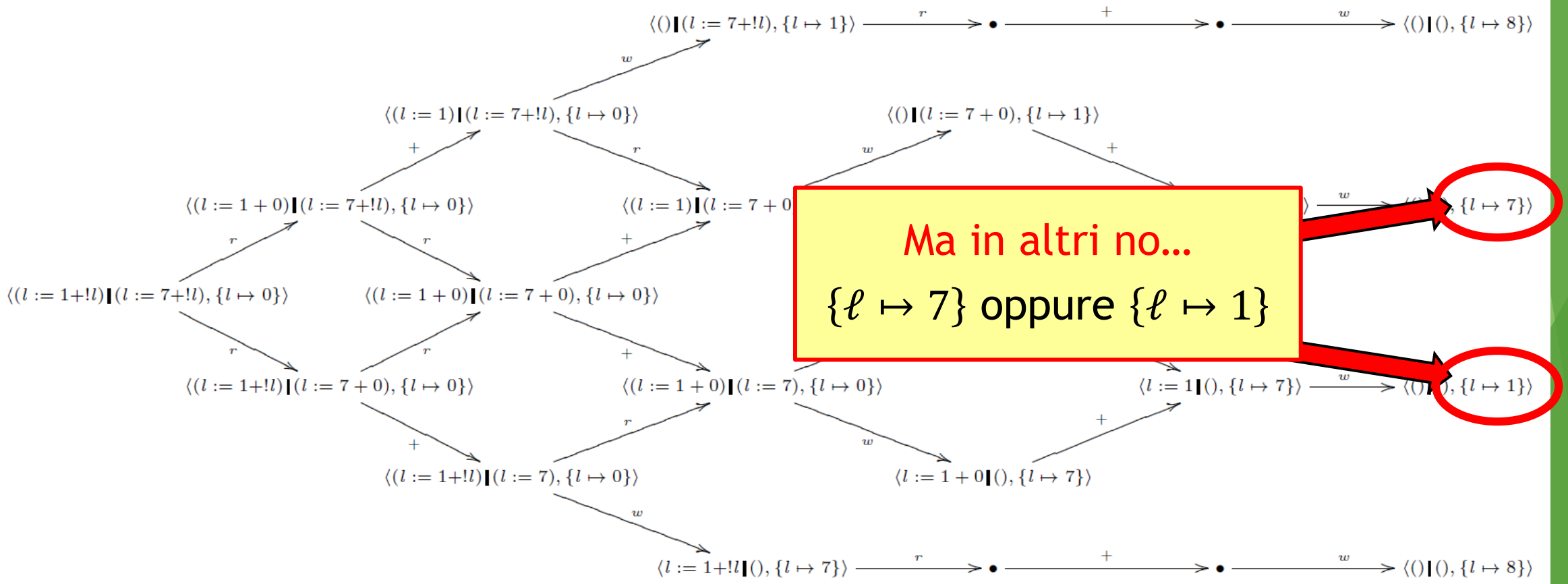
Esempio concorrente "problematico"

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



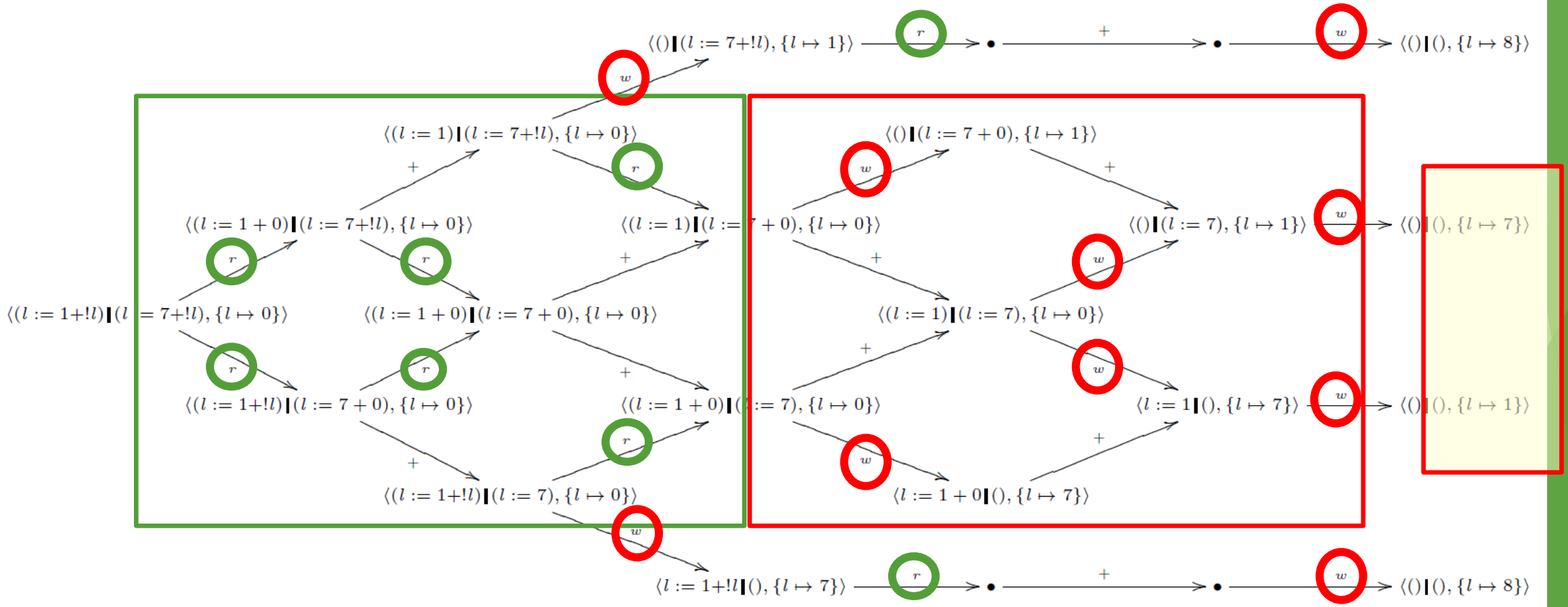
Esempio concorrente "problematico"

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

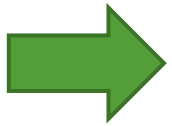
Regole semantiche aggiunte:

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



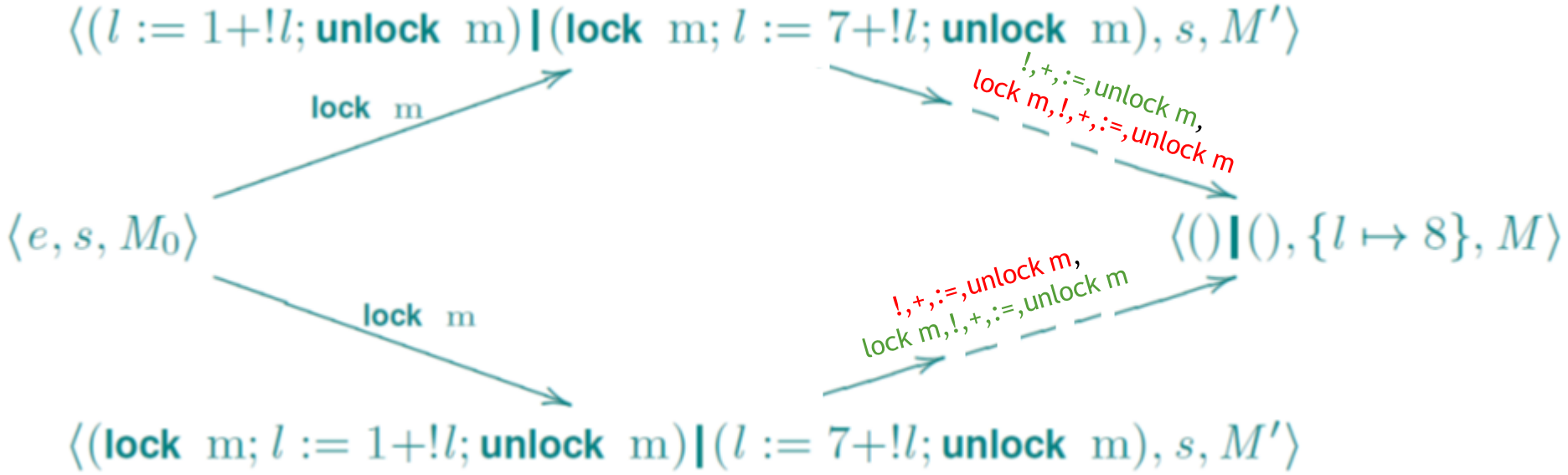
$$\frac{\langle e_1, s, M\rangle \to \langle e_1', s', M'\rangle}{\langle e_1;e_2, s, M\rangle \to \langle e_1';e_2\ , s', M'\rangle}$$

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:



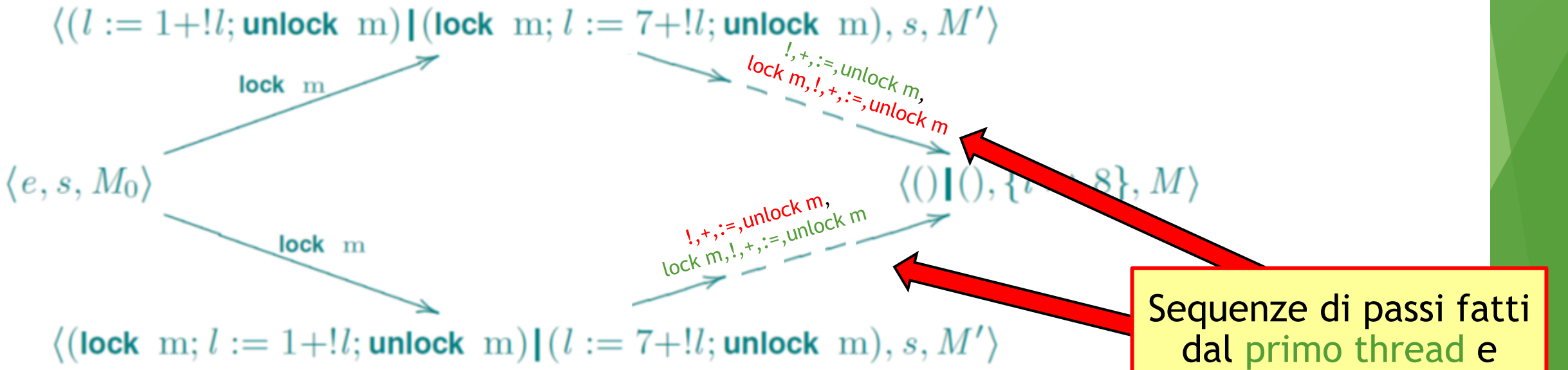
dove: $M' = M_0 + \{\mathrm{m} \mapsto \mathbf{true}\}$

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:



Sequenze di passi fatti dal primo thread e secondo thread

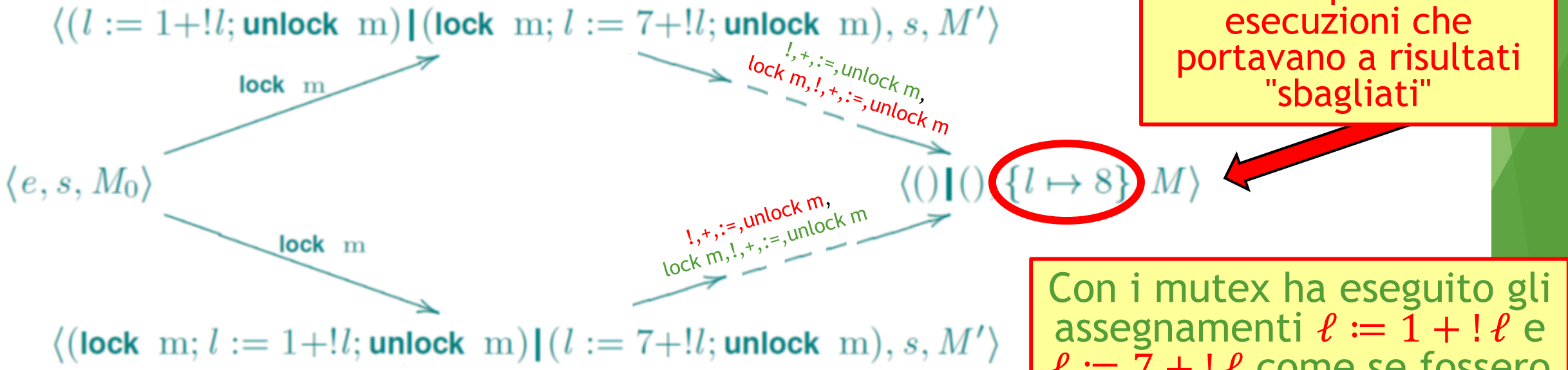
dove: $M' = M_0 + \{\mathrm{m} \mapsto \mathbf{true}\}$

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

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:

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

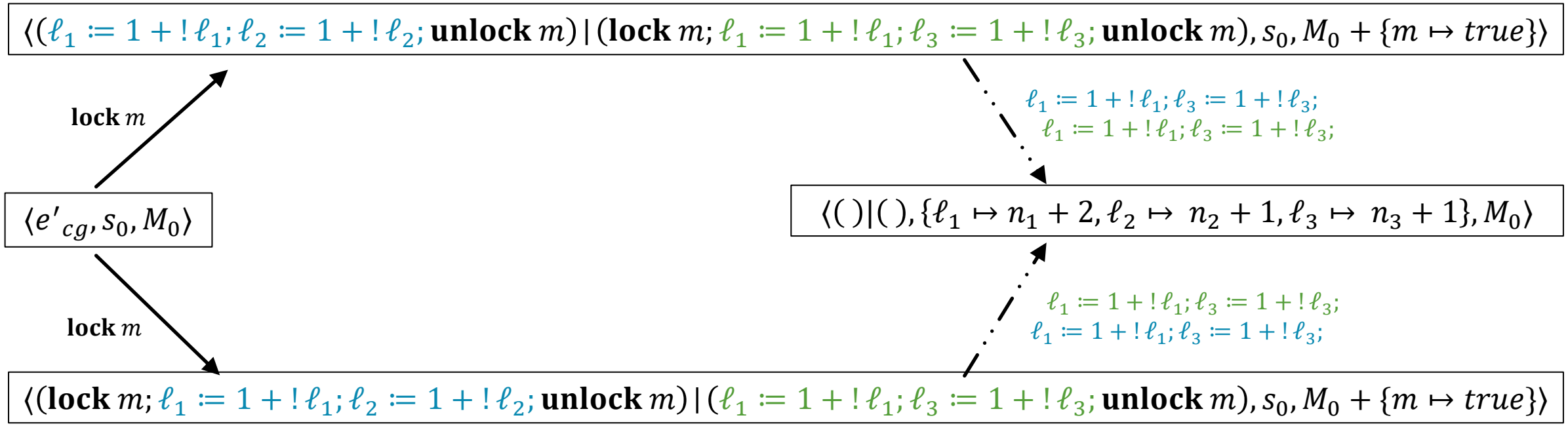
Esecuzione coarse-grained

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

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

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

Sistema di transizioni (possibili esecuzioni):



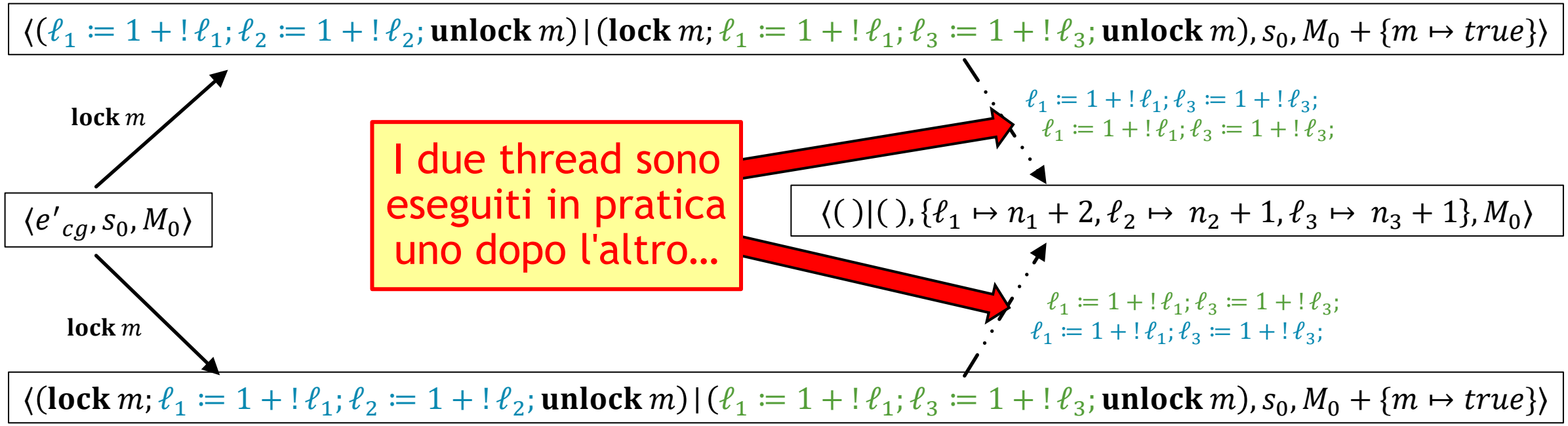
Esecuzione coarse-grained

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

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

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

Sistema di transizioni (possibili esecuzioni):



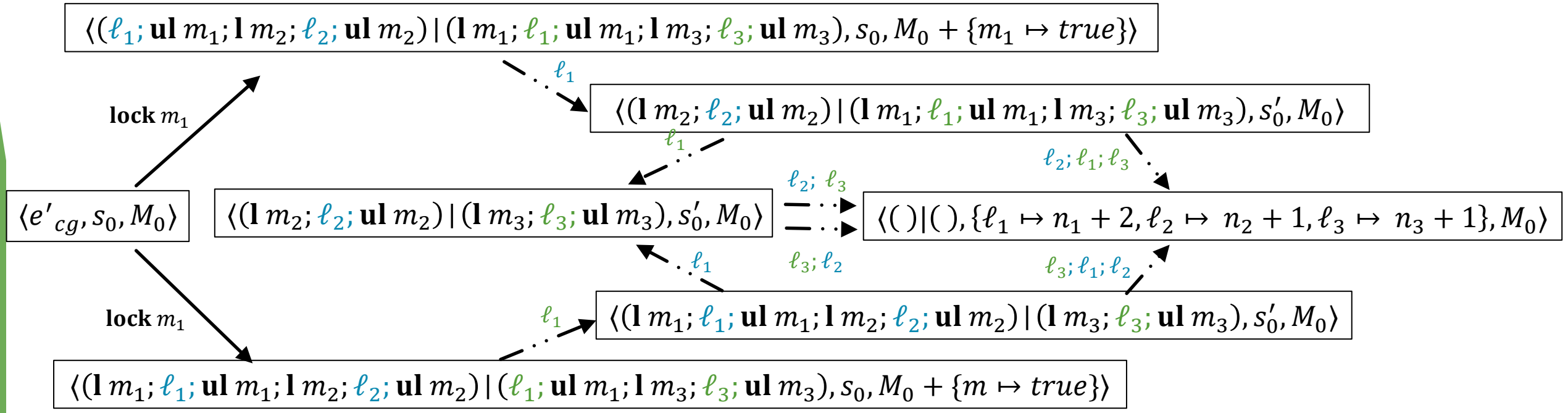
Esecuzione fine-grained

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

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

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

Sistema di transizioni (possibili esecuzioni):



Esecuzione fine-grained

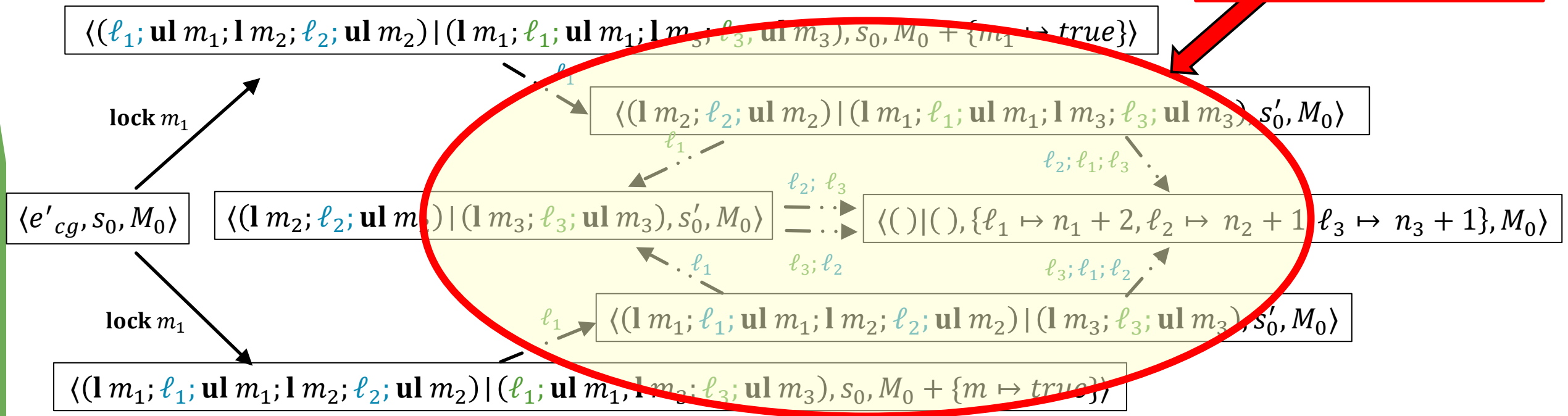
$$s_0 = \{\ell_1 \mapsto n_1, \ell_2 \mapsto n_2, \ell_3 \mapsto n_3\}$$

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

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

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?

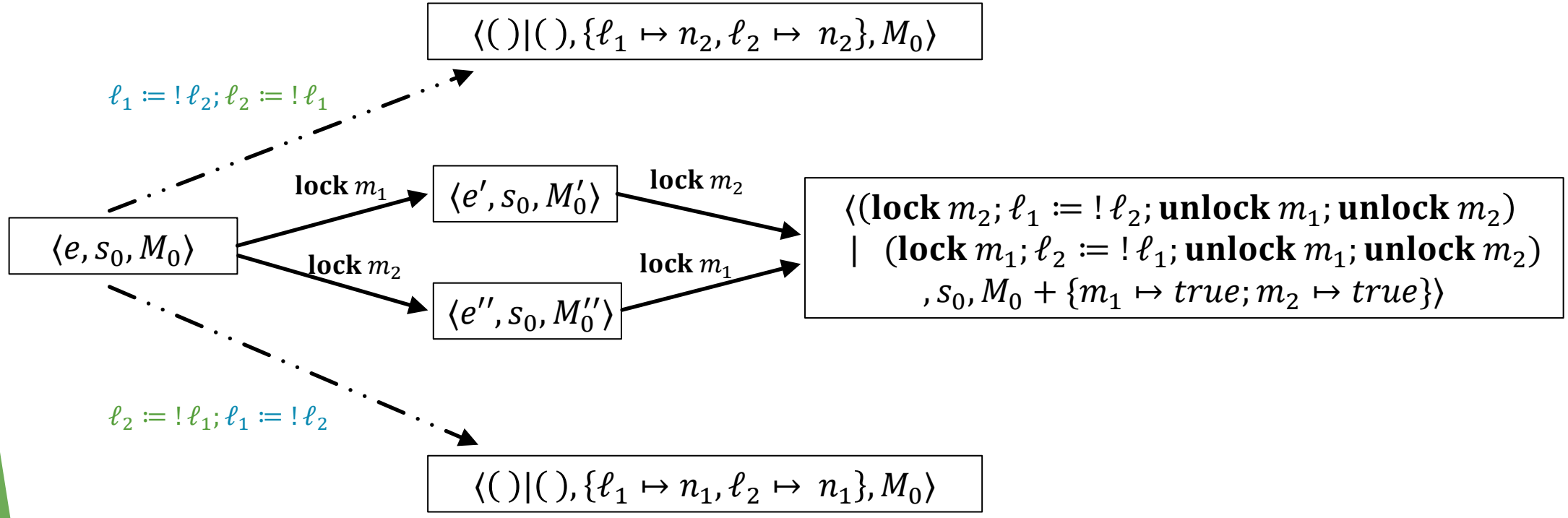
Uso pericoloso dei lock…

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

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

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

Sistema di transizioni (possibili esecuzioni):



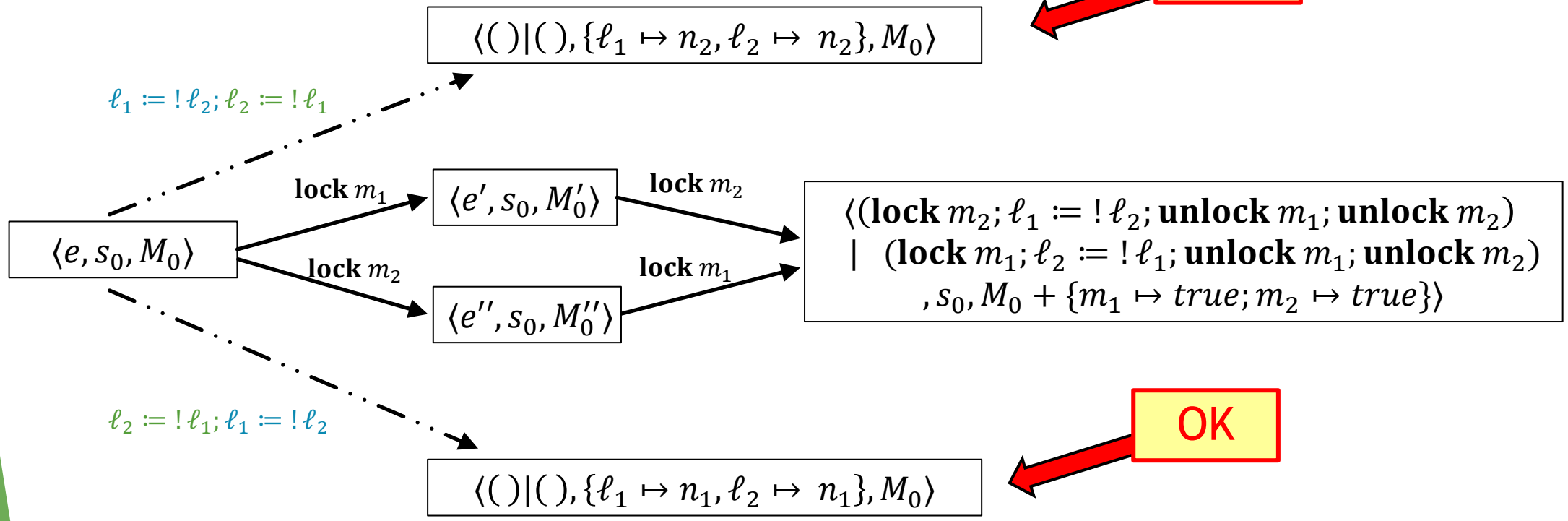
Uso pericoloso dei lock…

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

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

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

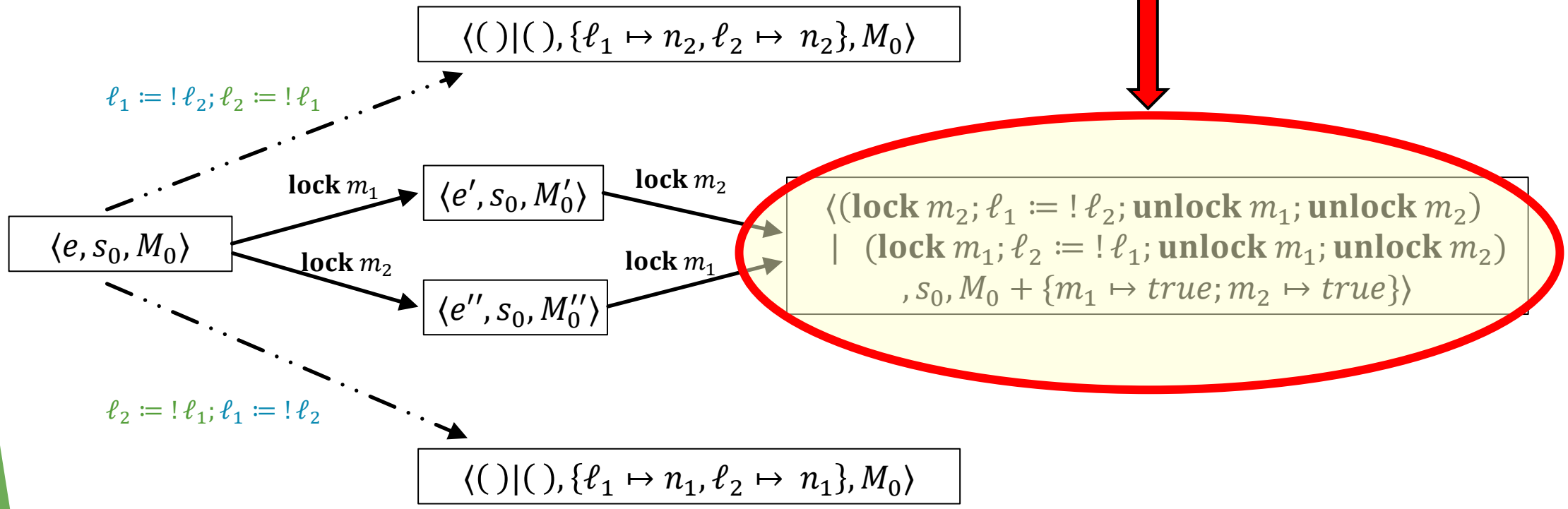
Sistema di transizioni (possibili esecuzioni):



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

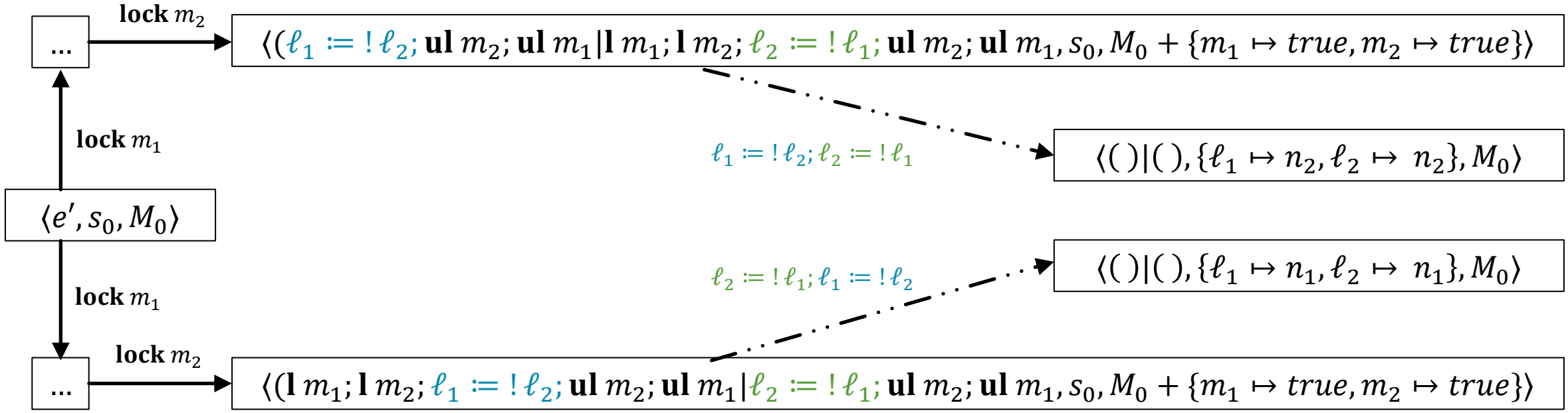
Esempio di deadlock prevention: 2-phase locking (2PL)

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

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

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

Sistema di transizioni (possibili esecuzioni):



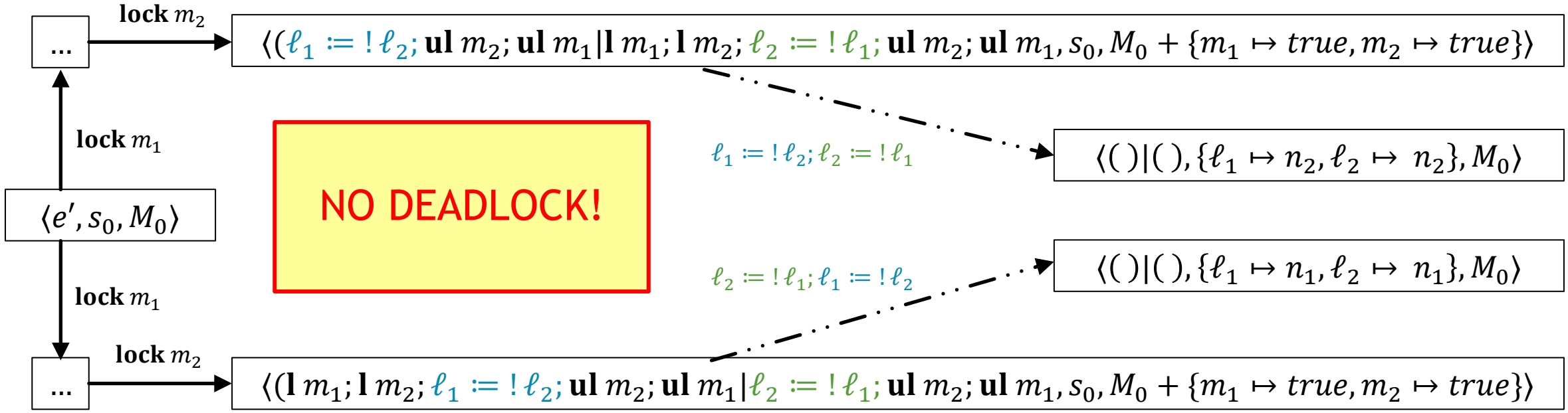
Esempio di deadlock prevention: 2-phase locking (2PL)

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

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

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

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