Concurrency

Massimo Merro

4 December 2017

Introduction

Our focus so far has been on the semantics of sequential computations.

However, many interesting systems are not sequential!

- hardware is intrisically parallel
- multiple-processor machines
- multi-threading (even on a single processor)
- networked machines
- cyber-physical systems
- IoT devices
- in general, concurrency can increase program performance by scheduling parallel independent tasks on multicore hardware, and can enable responsive user interfaces.

Challenges in concurrent systems

- the state-space of our systems become *larger*, with the *combinatorial explosion*; with n threads, each of which can be in only 2 states, the system has 2^n states!
- the state-space is not only larger but also more complex
- parallel components sharing resources should access them in mutual exclusion. If this is not done properly those components may suffer deadlock or starvation
- computations become nondeterministic (unless synchrony is imposed), as different threads operate at different speeds
- concurrency in programming might induce severe problems such as data races, i.e., concurrent access to shared data by different threads, with consequent unpredictable or erroneous behavior.

More challenges

- partial failures (of some process, of some device in a network, or some persistent storage device); need transaction mechanisms
- communication between different environments with different local resources (e.g. different local stores, or libraries); need consistency mechanisms;
- communication between administrative domains with partial trust (or, indeed not trust al all); protection against malicious attack
- dealing with contingent complexity (embedded historical accidents, etc).

On next slides

Theme: as for sequential languages seen up to now, but much more so. Concurrent languages are a complicated world.

Aim of this lecture: just to give you a taste of a how relatively simple semantics can be used to express some of the fine distinctions. Primarily

1. to boost your intuition on reasoning on concurrent systems
2. this can support rigorous proofs about crypto systems, cache-coherency protocols, database transactions, etc.

Our Goal: Define the simplest possible concurrent language and explore a few interesting issues.

A small concurrent language

linguaggio while originale e aggiungiamo il costrutto di composizione parallela

$$
\begin{array}{llll}
\textit{Booleans} & b \in \mathbb{B} & = & \{true, false\} \\
\textit{Integers} & n \in \mathbb{N} & = & \{\ldots, -1, 0, 1, \ldots\} \\
\textit{Locations} & l \in \mathbb{L} & = & \{\mathrm{l}, \mathrm{l}_0, \mathrm{l}_1, \mathrm{l}_2, \ldots\} \\
\textit{Operations} & op & ::= & + \mid \geq \\
\textit{Expressions} & e \in Exp & ::= & n \mid b \mid e\ op\ e \mid \text{if } e \text{ then } e \text{ else } e \\
 & & & \mid l := e \mid !l \mid skip \mid e; e \\
 & & & \mid \text{while } e \text{ do } e \mid e \parallel e \\
\textit{Types} & T & ::= & \text{int} \mid \text{bool} \mid \text{unit} \mid \text{proc} \\
 & T_{loc} & ::= & \text{intref}
\end{array}
$$

ho due thread che eseguono in parallelo e sono indipendenti

The construct $e \parallel e$ is called parallel composition.

Parallel composition: Our Design Choices

- threads don't return a value
- threads are anonymous, i.e. they don't have an identity
- termination of a thread cannot be directly observed withing a program
- processes, in general, are given by a pool of concurrent threads
- threads can't be killed externally.

Changes: Typing and operational semantics

single thread

multithread

$$\text{(T-sq1)}\ \frac{\Gamma \vdash e_1 : \mathsf{unit} \quad \Gamma \vdash e_2 : \mathsf{unit}}{\Gamma \vdash e_1; e_2 : \mathsf{unit}} \qquad \text{(T-sq2)}\ \frac{\Gamma \vdash e_1 : \mathsf{unit} \quad \Gamma \vdash e_2 : \mathsf{proc}}{\Gamma \vdash e_1; e_2 : \mathsf{proc}}$$

il programma diventa multithread

L'interprete introduce non determinismo perché non sa chi può andare avanti perché dipende da questioni di basso livello

$$\text{(T-par)}\ \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \parallel e_2 : \mathsf{proc}}\ T_1, T_2 \in \{\mathsf{unit}, \mathsf{proc}\}$$

multithread

$$\text{(par-L)}\ \frac{\langle e_1, s\rangle \twoheadrightarrow \langle e_1', s'\rangle}{\langle e_1 \parallel e_2,\ s\rangle \twoheadrightarrow \langle e_1' \parallel e_2,\ s'\rangle} \qquad \text{(par-R)}\ \frac{\langle e_2, s\rangle \twoheadrightarrow \langle e_2', s'\rangle}{\langle e_1 \parallel e_2,\ s\rangle \twoheadrightarrow \langle e_1 \parallel e_2',\ s'\rangle}$$

$$\text{(end-L)}\ \frac{-}{\langle skip \parallel e,\ s\rangle \twoheadrightarrow \langle e, s\rangle} \qquad \text{(end-R)}\ \frac{-}{\langle e \parallel skip,\ s\rangle \twoheadrightarrow \langle e, s\rangle}$$

- $\Gamma \vdash e : \mathsf{unit}$ entails e singlethreaded
- $\Gamma \vdash e : \mathsf{proc}$ entails e multithreaded

Il non determinismo permette di gestire sistemi complessi perciò se è gestito non è assolutamente un male

As in any concurrent language:

- threads execute asynchronously - the semantics allows any interleaving of the reductions of the threads
- all threads can read and write the shared memory
- As a consequence, the Determinacy property does not hold.

For instance:

$$\langle l := 1 \parallel l := 2, \{l \mapsto 0\}\rangle \twoheadrightarrow \langle \mathit{skip} \parallel l := 2, \{l \mapsto 1\}\rangle \twoheadrightarrow \langle \mathit{skip} \parallel \mathit{skip}, \{l \mapsto 2\}\rangle \twoheadrightarrow \langle \mathit{skip}, \{l \mapsto 2\}\rangle$$

But also

$$\langle l := 1 \parallel l := 2, \{l \mapsto 0\}\rangle \twoheadrightarrow \langle l := 1 \parallel \mathit{skip}, \{l \mapsto 2\}\rangle \twoheadrightarrow \langle \mathit{skip} \parallel \mathit{skip}, \{l \mapsto 1\}\rangle \twoheadrightarrow \langle \mathit{skip}, \{l \mapsto 1\}\rangle$$

Race conditions

- both 'assignments' and 'dereferencing' are atomic operations: in the previous configuration we can get a store where location l is associated to either 1 or 2. No strange combinations of them.
- However, in $(l := e) \parallel e'$ the semantic steps which are necessaty to evaluate e and e' can be interleaved
- So, what about the execution of program $(l := 1 + !l) \parallel (l := 7 + !l)$?
- In this case, we can get race conditions, i.e. the output can be something completely unexpected and inconsistent with the intentions of any thread!
- In particular, as decipted at pag. 97 of the notes, there are 3 possible final configurations for $\langle (l := 1 + !l) \parallel (l := 7 + !l), \{l \mapsto 0\}\rangle$:
  1. $\langle \mathit{skip}, \{l \mapsto 1\}\rangle$
  2. $\langle \mathit{skip}, \{l \mapsto 7\}\rangle$
  3. $\langle \mathit{skip}, \{l \mapsto 8\}\rangle$
- (1) and (2) are due to "interferences" while executing the assignments; only (3) corresponds to some correct scheduling!

Morals

- There are too many possible results
- Actually, all the possible executions give rise to to a combinatorial explosion of states
- Drawing state-space diagrams, as done at pag. 97 of Sewell's notes, works only for very little examples: we need better techniques to analyze our concurrent programs!
- Almost centainly you (as the programmer) didn't want all those 3 outcomes to be possible - need better idioms to or constructs for programming.

How do we get anything coherent done?

- need some way(s) to synchronize between threads, so can enforce mutual exclusion for shared data
- Think of Lamport's "Bakery" algorithm for concurrent and distributed systems. Can you code that in our small concurrent language? If not, what would you need in the language?
- though you can depend on built-in support from the scheduler, e.g. mutexes or condition variable (or, at the lower level, **tas**, test-and-set, or **cas**, compare-and-set).

Adding primitives mutexes in the language

Mutex names $m \in \mathbb{M} = \{\mathrm{m}, \mathrm{m}_1, \dots\}$

Configurations $\langle e, s, \mu \rangle$, where $\mu : \mathbb{M} \to \mathbb{B}$ is the mutex state

Expressions $e \in Exp \dots \mid e \parallel e \mid \mathsf{lock}\ m \mid \mathsf{unlock}\ m$

Typing:

$$(\text{T-lock})\ \frac{-}{\Gamma \vdash \mathsf{lock}\ m : \mathsf{unit}} \qquad (\text{T-unlock})\ \frac{-}{\Gamma \vdash \mathsf{unlock}\ m : \mathsf{unit}}$$

Operational semantics:

$$(\text{lock})\ \frac{-}{\langle \mathsf{lock}\ m, s, \mu \rangle \rightarrow \langle skip, s, \mu[m \mapsto true] \rangle} \quad \text{if } \neg\mu(m)$$

$$(\text{unlock})\ \frac{-}{\langle \mathsf{unlock}\ m, s, \mu \rangle \rightarrow \langle skip, s, \mu[m \mapsto false] \rangle}$$

... and adapt all the other rules to extended configurations $\langle e, s, \mu \rangle$.

Using a Mutex

To avoid race conditions, we can rewrite the previous program as follows:

$$Prg \stackrel{\text{def}}{=} (\text{lock } \mathrm{m}; \mathrm{l} := 1 + !\mathrm{l}; \text{unlock } \mathrm{m}) \parallel (\text{lock } \mathrm{m}; \mathrm{l} := 7 + !\mathrm{l}; \text{unlock } \mathrm{m})$$

Let $\langle Prg, s_0, \mu_0 \rangle$ be a configuration such that $s_0 = \{\mathrm{l} \mapsto 0\}$ and μ_0 returns *false* for any mutex name. Then, for all possible executions traces of the configuration $\langle Prg, s_0, \mu_0 \rangle$ we will always have

$$\langle Prg, s_0, \mu_0 \rangle \twoheadrightarrow^* \langle skip, \{\mathrm{l} \mapsto 8\}, \mu_0 \rangle$$

No other final configurations are possible!

The two assignments will be executed one after the other in mutual exclusion.

Note that the two assignments *commute*, so we end up in the same final state whichever got the lock first.

Deadlocks

The construct lock *m* can block if the the mutex *m* has already been locked by another thread. So, if we use (at least) two mutexes we can easily deadlock!

Consider

$$
\begin{array}{rll}
e = & & (\mathsf{lock}\ \mathrm{m}_1; \mathsf{lock}\ \mathrm{m}_2; \mathrm{l}_1 := !\mathrm{l}_2; \mathsf{unlock}\ \mathrm{m}_1; \mathsf{unlock}\ \mathrm{m}_2) \\
 & \| & (\mathsf{lock}\ \mathrm{m}_2; \mathsf{lock}\ \mathrm{m}_1; \mathrm{l}_2 := !\mathrm{l}_1; \mathsf{unlock}\ \mathrm{m}_2; \mathsf{unlock}\ \mathrm{m}_1)
\end{array}
$$

... and we don't want deadlocks!

Language Properties

- Obviously, we don't have Determinacy anymore
- Type preservation is still valid
- Typing and type inferences is scarcely changed
- Very fancy type systems can be used to enforce locking disciplines
- Progress in general is not valid unless we adopt a type system to enforce a locking discipline. In that case, we would have deadlock-freedom for free. This has an influence on our notions of semantic equivalence.

Semantic equivalences on concurrent programs

Since deadlocking processes are not ruled out anymore by our type system, we have to revisit our semantic equivalences

Let's amend the typed equivalences seen for sequential computations.

Trace equivalence $\simeq_\Gamma$

$e_1 \;\simeq_\Gamma\; e_2$ iff for all mutex states μ and all stores s, s.t. $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$, we have $\Gamma \vdash e_1 : T_1$, $\Gamma \vdash e_2 : T_2$, $T_1, T_2 \in \{\mathrm{unit}, \mathrm{proc}\}$, and

- $\langle e_1, s, \mu\rangle \longrightarrow^* \langle e_1', s', \mu'\rangle$ implies $\exists e_2'. \langle e_2, s, \mu\rangle \longrightarrow^* \langle e_2', s', \mu'\rangle$
- $\langle e_2, s, \mu\rangle \longrightarrow^* \langle e_2', s', \mu'\rangle$ implies $\exists e_1'. \langle e_1, s, \mu\rangle \longrightarrow^* \langle e_1', s', \mu'\rangle$.

Notice that now we consider also partial traces and not only those leading to final configurations.

Example (1)

$$P_1 = ((\mathrm{lock\ m}; \mathrm{l} := 3) \parallel (\mathrm{lock\ m}; \mathrm{l} := 4))$$
$$Q_1 = \mathrm{lock\ m}; (\mathrm{l} := 3 \parallel \mathrm{l} := 4)$$

- Is $P_1 \simeq_\Gamma Q_1$, for $\Gamma = \{\mathrm{l} : \mathsf{intref}\}$? Yes, it is!
- Is $C[P_1] \simeq_{\Gamma'} C[Q_1]$, for any well-typed context $C[\cdot]$? No, it isn't!.
- Consider the context $C[\cdot]$ defined as follows:

$$[\cdot] \parallel (\mathrm{x}_1 := !\mathrm{l}; \mathrm{x}_2 := !\mathrm{l}; \mathrm{if}\ !\mathrm{x}_2 = !\mathrm{x}_1 + 1\ \mathrm{then}\ \mathrm{r} := 1\ \mathrm{else}\ \mathrm{r} := 0)$$

- Then $\langle C[Q_1], s, \mu \rangle \twoheadrightarrow^* \langle skip, s'\ \mu' \rangle$, with $s(\mathrm{l}) = s(\mathrm{r}) = 0$, $\mu(\mathrm{m}) = false$, $s'(\mathrm{l}) = 4$, $s'(\mathrm{r}) = 1$ and $\mu'(m) = true$.
- But, there is no trace $\langle C[P_1], s, \mu \rangle \twoheadrightarrow^* \langle \ldots, s'', \mu'' \rangle$, with $s' = s''$ and $\mu' = \mu''$! This is because P_1 can "touch" location l only once!
- However, $C[\cdot]$ is not "fair" because it does not acquire the mutex before accessing location l. Any "fair" distinguishing context?

Example (2)

Suppose we can type the following programs:

$$
\begin{array}{lcl}
P_2 & = & ((\mathrm{lock\ m}; \mathrm{l} := 3; \mathrm{unlock\ m}) \parallel (\mathrm{lock\ m}; \mathrm{l} := 4; \mathrm{unlock\ m})) \\
Q_2 & = & \mathrm{lock\ m}; (\mathrm{l} := 3 \parallel \mathrm{unlock\ m}; \mathrm{lock\ m} \parallel \mathrm{l} := 4); \mathrm{unlock\ m} \\
R_2 & = & \mathrm{lock\ m}; (\mathrm{l} := 3 \parallel \mathrm{unlock\ m} \parallel \mathrm{lock\ m} \parallel \mathrm{l} := 4); \mathrm{unlock\ m}
\end{array}
$$

In these 3 programs the critical assignments to l are fully locked.

- Is $P_2 \simeq_\Gamma Q_2 \simeq_\Gamma R_2$, for $\Gamma = \{\mathrm{l}_0 : \mathsf{intref}, \mathrm{l} : \mathsf{intref}\}$? Yes, it is.
- Is $C[P_2] \simeq_\Gamma C[Q_2]$, for any "fair" context $C[\cdot]$? No, it isn't!
- Consider the "fair" context $C[\cdot]$ defined as follows:

$$[\cdot] \parallel (\mathrm{lock\ m}; \mathrm{x}_1{:=}!\mathrm{l}; \mathrm{x}_2{:=}!\mathrm{l}; (\mathrm{if}\ !\mathrm{x}_2 = !\mathrm{x}_1{+}1\ \mathrm{then}\ \mathrm{r}{:=}1\ \mathrm{else}\ \mathrm{r}{:=}0); \mathrm{unlock\ m})$$

- Then $\langle C[Q_2], s, \mu\rangle \twoheadrightarrow^* \langle \ldots, s'\ \mu'\rangle$, with $s(\mathrm{l}){=}s(\mathrm{r}){=}0$, $\mu(\mathrm{m}){=}\mathit{false}$, $s'(\mathrm{l}) = 4$, $s'(\mathrm{r}){=}1$ and $\mu'(m){=}\mathit{true}$.
- But there is no trace s.t. $\langle C[P_2], s, \mu\rangle \twoheadrightarrow^* \langle \ldots, s', \mu'\rangle$.

So...

- It is not considering "fair" contexts that we fix the problem!
- What is the problem? $\simeq_\Gamma$ is not preserved by parallel contexts!
- Why? Because trace equivalence forgets about intermediate states!

Moral: Parallel contexts have a stronger distinghishing power because they have more chances to create interferences.

- That's why it is much more difficult to write correct concurrent programs: when you add parallel threads a correct sequential program may go wrong!

Trace Congruence: a much finer semantic equivalence

Trace congruence $\cong_\Gamma$

Define $e_1 \;\cong_\Gamma\; e_2$ to hold iff for all mutex states μ and all stores s, such that $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$, we have $\Gamma \vdash e_1 : T_1$, $\Gamma \vdash e_2 : T_2$, $T_1, T_2 \in \{\mathrm{unit}, \mathrm{proc}\}$ and

- $\langle e_1, s, \mu \rangle \twoheadrightarrow^* \langle e_1', s', \mu' \rangle$ implies $\exists e_2'. \langle e_2, s, \mu \rangle \twoheadrightarrow^* \langle e_2', s', \mu' \rangle$
- $\langle e_2, s, \mu \rangle \twoheadrightarrow^* \langle e_2', s', \mu' \rangle$ implies $\exists e_1'. \langle e_1, s, \mu \rangle \twoheadrightarrow^* \langle e_1', s', \mu' \rangle$
- if $e_1 \cong_\Gamma e_2$ then *for any expression* e such that $\Gamma' \vdash e_1 \parallel e : \mathrm{proc}$ and $\Gamma' \vdash e_2 \parallel e : \mathrm{proc}$, for some Γ', then $e_1 \parallel e \cong_{\Gamma'} e_2 \parallel e$.

By definition, the relation $\cong_\Gamma$ is preserved by parallel contexts!

... and

$$P_1 \;\ncong_\Gamma\; Q_1$$

$$P_2 \;\ncong_\Gamma\; Q_2$$

What about bi-similarity for concurrent programs?

We adapt the definitions to the current setting with mutexes:

Similarity

We say that e_1 is simulated by e_2, written $e_1 \sqsubseteq_\Gamma e_2$, iff

- $\Gamma \vdash e_1 : T_1$ and $\Gamma \vdash e_2 : T_2$, with $T_1, T_2 \in \{\text{unit}, \text{proc}\}$
- for any μ and s with $\text{dom}(\Gamma) \subseteq \text{dom}(s)$, if $\langle e_1, s, \mu \rangle \rightarrow \langle e_1', s', \mu' \rangle$ then there is e_2' such that $\langle e_2, s, \mu \rangle \rightarrow^* \langle e_2', s' \mu' \rangle$, with $e_1' \sqsubseteq_\Gamma e_2'$.

Bisimilarity

→ è più forte

We say that e_1 is bisimilar to e_2, written $e_1 \approx_\Gamma e_2$, iff

- $\Gamma \vdash e_1 : T_1$ and $\Gamma \vdash e_2 : T_2$, with $T_1, T_2 \in \{\text{unit}, \text{proc}\}$
- for any μ and s with $\text{dom}(\Gamma) \subseteq \text{dom}(s)$, if $\langle e_1, s, \mu \rangle \rightarrow \langle e_1', s', \mu' \rangle$ then there is e_2' such that $\langle e_2, s, \mu \rangle \rightarrow^* \langle e_2', s', \mu' \rangle$, with $e_1' \approx_\Gamma e_2'$
- for any μ and s with $\text{dom}(\Gamma) \subseteq \text{dom}(s)$, if $\langle e_2, s, \mu \rangle \rightarrow \langle e_2', s', \mu' \rangle$ then there is e_1' such that $\langle e_1, s, \mu \rangle \rightarrow^* \langle e_1', s', \mu' \rangle$, with $e_1' \approx_\Gamma e_2'$.

$$P \approx_t Q \quad \text{then} \quad P \downarrow^* \langle \dots, s', \mu' \rangle \qquad Q \downarrow^* \langle \dots, s', \mu' \rangle$$

$$C[P] \not\approx_t C[Q]$$

$$P \approx_s Q \quad \text{then} \quad P \downarrow^* \langle P', s', \mu' \rangle \leq_t \langle Q', s', \mu' \rangle \;{}^* \!\!\uparrow Q$$

Again

$$\boxed{C[P] \approx C[Q]}$$

$$P \approx Q \Rightarrow \begin{cases} P \leq Q \\ \wedge \\ P \geq Q \end{cases}$$

Now, if you consider the processes of the previous examples:

$$
\begin{array}{ccc}
P_1 & \sqsubseteq_\Gamma & Q_1 \\
Q_1 & \not\sqsubseteq_\Gamma & P_1 \\
P_2 & \sqsubseteq_\Gamma & Q_2 \\
Q_2 & \not\sqsubseteq_\Gamma & P_2 \\
Q_2 & \approx_\Gamma & R_2
\end{array}
$$

Unlike $\simeq_\Gamma$, both $\sqsubseteq_\Gamma$ and $\approx_\Gamma$ can observe changes at intermediate states!

In general, $P \sqsubseteq_\Gamma Q$ and $Q \sqsubseteq_\Gamma P$ does not imply $P \approx_\Gamma Q$! Can you find two programs P and Q where this happens?

Is the relation $\sqsubseteq_\Gamma$ a congruence? Yes, it is!.

Why? Because $\sqsubseteq_\Gamma$ is much sharper when observing processes.

On the power of bi-similarity

Similarity and Bisimilarity are preserved by parallel contexts

- If $e_1 \sqsubseteq_\Gamma e_2$ then *for any expression* e, such that $\Gamma' \vdash e_1 \parallel e : \mathsf{proc}$ and $\Gamma' \vdash e_2 \parallel e : \mathsf{proc}$, for some Γ', it holds that

$$e_1 \parallel e \;\sqsubseteq_\Gamma\; e_2 \parallel e \;\; .$$

- If $e_1 \approx_\Gamma e_2$ then *for any expression* e, such that $\Gamma' \vdash e_1 \parallel e : \mathsf{proc}$ and $\Gamma' \vdash e_2 \parallel e : \mathsf{proc}$, for some Γ', it holds that

$$e_1 \parallel e \;\approx_\Gamma\; e_2 \parallel e \;\; .$$

$\boxed{M \| A \subseteq M}$

attacco tollerato
non influenza il
sistema

Conditional critical regions

- We have seen that communication between parallel threads is via the store
- In concurrent programs it is very difficult to limit interferences on it
- Many real concurrent programming languages have constructs for alleviating these problems: *semaphores*, *locks*, *critical regions*, etc
- We have seen how to enrich our language with a simple form of locks
- Here we examine a higher-level construct for conditional critical regions

costrutto di alto livello che vuole permettere l'esecuzione di un processo e_2 in maniera atomica

$$\text{await } e_1 \text{ protect } e_2 \text{ end}$$

- The intuition is that this command may only be executed when the boolean expression e_1 is true, and the entire command e_2 is to be executed to completion without interruption or interference.

- For example consider the program:

$$Prg_1 \stackrel{\text{def}}{=} \mathrm{l} := 0 \parallel (\text{await } !\mathrm{l} = 0 \text{ protect } \mathrm{l} := 1; \mathrm{l} := !\mathrm{l} + 1 \text{ end})$$

- This is a deterministic program; if it is executed in a state s, with $s(\mathrm{l}) \neq 0$, then it will terminate and the only possible terminal state is $s[\mathrm{l} \mapsto 2]$.
- As another example consider the more involved program:

$$\begin{aligned} Prg_2 \stackrel{\text{def}}{=} \; & (\text{await true protect } \mathrm{l}_1 := 1; \mathrm{l}_1 := !\mathrm{l}_0 + 1 \text{ end}) \\ & \parallel \\ & (\text{await true protect } \mathrm{l}_0 := 2; \mathrm{l}_0 := !\mathrm{l}_1 + 1 \text{ end}) \end{aligned}$$

- The two guards, set to true, are vacuous, so which protected command is executed first is chosen non-deterministically.

Formally...

await false do skip; e_1

Language:

Configurations $\langle e, s \rangle$, as before

Expressions $e \in Exp ::= \ldots \mid e \parallel e \mid \texttt{await } e \texttt{ protect } e \texttt{ end}$

Typing:

$$(\text{T-await}) \ \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \text{await } e_1 \text{ protect } e_2 \text{ end} : \text{unit}}$$

Operational semantics:

$$(\text{await}) \ \frac{\langle e_1, s \rangle \twoheadrightarrow^* \langle \mathit{true}, s' \rangle \quad \langle e_2, s' \rangle \twoheadrightarrow^* \langle \mathit{skip}, s'' \rangle}{\langle \text{await } e_1 \text{ protect } e_2 \text{ end}, s \rangle \twoheadrightarrow \langle \mathit{skip}, s'' \rangle}$$

This is a kind of test-and-set command: whenever the guard e_1 evaluates to true the command e_2 can be executed atomically, in just one step!

It is easy to see that

$$\text{await } \mathit{true} \text{ protect } (\mathrm{l} := !\mathrm{l} + 1; \mathrm{l} := !\mathrm{l} - 1) \text{ end}$$

$$\approx_\Gamma$$

$$\text{await } \mathit{false} \text{ protect } (\mathrm{l} := !\mathrm{l} + 1; \mathrm{l} := !\mathrm{l} - 1) \text{ end}$$

$$\approx_\Gamma$$

$$\text{await } e \text{ protect } (\mathrm{l} := !\mathrm{l} + 1; \mathrm{l} := !\mathrm{l} - 1) \text{ end}$$

$$\approx_\Gamma$$

$$\mathit{skip}$$

for any expression e such that

- $\Gamma \vdash e : \text{bool}$
- e does not modify the store.

Example

Let us consider the following programs:

$$P_4 \stackrel{\text{def}}{=} l_0 := 0; (\text{await } !l_0 = 0 \text{ protect } (l := 1; l_0 := 1) \text{ end}) \parallel (\text{await } !l_0 = 0 \text{ protect } (l := 0; l_0 := 1) \text{ end})$$

$$Q_4 \stackrel{\text{def}}{=} l_0 := 0; (l := 0; l_0 := 1 \parallel l := 1; l_0 := 1)$$

Supponendo di poter tipare il seguente processo:

$$R_4 \stackrel{\text{def}}{=} l_0 := 0; (\text{await } !l_0 = 0 \text{ protect } (l := 0; l_0 := 1 \parallel l := 1; l_0 := 1) \text{ end})$$

P_4 simula Q_4

- $P_4 \sqsubseteq_\Gamma Q_4$
- $Q_4 \not\sqsubseteq_\Gamma P_4$ → Q_4 non simula P_4
- $P_4 \approx_\Gamma R_4$ → bisimili

Nondeterministic choice

Let us suppose to enrich our language with the following construct:

$$\begin{array}{lll} \textit{Configurations} & \langle e, s\rangle, \text{ as before} & \\ \textit{Expressions} & e \in \textit{Exp} \ ::= \ \ldots & \mid \ e + e \end{array}$$

scelta non deterministica

Typing:

$$\text{(T-choice)} \ \frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash e_1 + e_2 : \text{unit}}$$

Operational semantics:

$$\text{(ChoiceL)} \ \frac{\langle e_1, s\rangle \twoheadrightarrow \langle e_1', s'\rangle}{\langle e_1 + e_2, s\rangle \twoheadrightarrow \langle e_1', s'\rangle}$$

e1 prende il sopravvento, e2 viene scartato

$$\text{(ChoiceR)} \ \frac{\langle e_2, s\rangle \twoheadrightarrow \langle e_2', s'\rangle}{\langle e_1 + e_2, s\rangle \twoheadrightarrow \langle e_2', s'\rangle}$$

e2 prende il sopravvento e1 viene scartato

This construct chooses nondeterministically one branch or the other; once a branch is chosen the other one is discarded!

$skip + \ell := 1 ; \ell := 2 \xrightarrow{\ell := 1} \ell := 2$

qui non vado mai in skip
perché é un processo morto mentre
l'altro puó tranquillamente proseguire

$e_1 + e_2 \xrightarrow{testa} e_1$

$e_1 + e_2 \xrightarrow{croce} e_2$

in questo caso e con questa
implementazione non avrei il medesimo
comportamento

poiché devo valutare se entrambi i
processi possono proseguire, in quel
caso tiro la monetina

$skip + skip \xrightarrow{deadlock}$

$skip \sqsubseteq_R P$

skip é simulato da
qualsiasi processo
basta che stia
fermo visto che non
fa nulla

True and false algebraic laws

bisimile

1. $e + e \approx_\Gamma e$
2. $e \approx_\Gamma skip; e$
3. $e_1 + e_2 \sqsubseteq_\Gamma (skip; e_1) + e_2$
4. $e_1 + e_2 \sqsupseteq_\Gamma (skip; e_1) + e_2$
5. $e_1 + e_2 \not\approx_\Gamma (skip; e_1) + e_2$
6. $e_1 + e_2 \sqsubseteq_\Gamma (skip; e_1) + (skip; e_2)$
7. $e_1 + e_2 \sqsupseteq_\Gamma (skip; e_1) + (skip; e_2)$
8. $e_1 + e_2 \not\approx_\Gamma (skip; e_1) + (skip; e_2)$
9. $e + e \approx_\Gamma (skip; e) + e$
10. $e + e \approx_\Gamma (skip; e) + (skip; e)$

(1) $e + e \overset{?}{\dot\approx} e$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1'} \qquad \frac{e_2 \to e_2'}{e_1 + e_2 \to e_2'}$$

$$\begin{tikzcd}
e + e \arrow[d] & \overset{?}{\dot\approx} & e \arrow[d] \\
e' & & e'
\end{tikzcd}$$

$e \to e'$

perché $e \to e'$

(2)

$$\begin{tikzcd}
e \arrow[d] \arrow[r, phantom, "\overset{?}{\approx}"] & skip;e \arrow[d, "skip"] \\
e' \arrow[dr, dash, "\text{identità}"'] & e \arrow[d] \\
 & e'
\end{tikzcd}$$

bisimulazione dice che si può rispondere con uno o più passi

Allora $e \sqsubseteq_r skip;e$

$$\begin{tikzcd}
e \arrow[d, "*"] \arrow[r, phantom, "\overset{?}{\approx}"] & skip;e \arrow[d] \\
e \arrow[r, dash, "\text{identità}"'] & e
\end{tikzcd}$$

$e \sqsupseteq skip;e$

Ho $e \approx_r skip;e$

③ $\overset{P}{\overset{\shortparallel}{e_1+e_2}} \sqsubseteq \overbrace{skip; e_1+e_2}^{Q}$

Provo a costruire una relazione di SIMULAZIONE $\mathcal{R} = \{(P,Q), \ldots\ldots$

① Analizziamo i passi possibili nella 1ª coppia (da sx vs dx)

1.1 Se $P = e_1+e_2 \to e_1' \equiv P'$ $e_1 \to e_1' \equiv P'$ $e_1 \to e_1'$ e ho applicato la (choice L). Come risponde Q?

Allora $Q \equiv skip; e_1+e_2 \to e_1 \to e_1' \equiv Q'$ con $(P',Q) \in \mathcal{R}$

1.2 Se $P \equiv e_1+e_2 \to e_2' \equiv P'$ perché $e_2 \to e_2'$ e ho applicato la (choice R). Cosa risponde Q?

Allora $Q \equiv skip$ $e_1+e_2 \to e_2' \equiv Q'$, con $(P', Q') \in \mathcal{R}$

④ $\overset{P}{\overset{\shortparallel}{e_1+e_2}} \sqsupseteq \overbrace{skip; e_1+e_2}^{Q}$ $\mathcal{R} = \{(P,Q), \ldots\ldots \quad \} \cup Id$

① Analizziamo la 1ª coppia

1) Se $Q \equiv skip; e_1+e_2 \to e_1 \equiv P'$, per una applicazione delle regole. P può simulare Q?
Allora $P \to^{*} P \equiv P'$ con $(Q', P') \in \mathcal{R}$

2) Se $Q \equiv skip; e_1+e_2 \to e_2'$, applicando la (choice R). Ma allora $Q = e_1+e_2 \to e_2' \equiv Q$, applicando la (choice R) con $(Q', P') \in Id \subseteq \mathcal{R}$

$e_1 \equiv \ell := 1$
$e_2 \equiv \ell := 2$

② Analizziamo la 2ª coppia

2.1 Se $e_1 \to e_1'$, per qualche e_1'. Allora $e_1+e_2 \to e_1'$, applicando la (choice L) con $(e_1', e_1') \in Id \subseteq \mathcal{R}$

⑤

$$P = e_1 + e_2 \;\overset{\sqsubseteq}{\underset{\approx}{\sqsupseteq}}\; skip; e_1 + skip; e_2 = Q$$

$$R = \{ (Q,P).\ (e_1;\ e_1+e_2) \} \cup Id$$

2) $e_1 + e_2 \to^* e_1 + e_2 \overset{?}{\approx} e_1$, con $Q \xrightarrow{skip(1)} e_1$, $e_1 \not\to$

3) $e_1 + e_2 \to e_2'$

$m = 3$

$e_1 \equiv l := 5$

$e_2 \equiv m := 3$

$$e_1 + e_2 \overset{?}{\sqsupseteq} skip, e_1 + skip; e_2 = Q$$

1) Analizziamo 1ª coppia di R

1.1) Sia $Q \to e_1$, per la (choice L). Allora $P \to^* P \equiv e_1 + e_2$, con $(e_1, e_1+e_2) \in R$

1.2) Sia $Q \to e_2$, per la (choice R). Allora $P \to^* P' \equiv e_1 + e_2$, con $(e_2, e_1+e_2) \in R$

2) Analizziamo la 2ª coppia

$\hookrightarrow$ $e_1 \to e_1$, ma allora $e_1 + e_2 \to e_1'$ (choice L) con $(e_1', e_1') \in Id \subseteq R$

3) Analizziamo la terza coppia

finisce nel caso precedente

non genera altre coppie in R perché non ricade nell'identità

$$e_1 + e_2 \overset{?}{\sqsubseteq} skip, e_1 + skip; e_2 = Q$$

$$R = \{ (P,Q). \qquad \} \cup Id$$

1) Analizziamo 1ª coppia di R

Example: When execution order is important

Consider the following algebraic law:

$$\mathrm{l} := 1 \parallel \mathrm{m} := 2 \quad \approx_\Gamma \quad (\mathrm{l} := 1; \mathrm{m} := 2) + (\mathrm{m} := 2; \mathrm{l} := 1)$$

Can we generalise this law as follows?

$$e_1 \parallel e_2 \quad \approx_\Gamma \quad (e_1; e_2) + (e_2; e_1)$$

e_1 l=0 ; m=2

e_2 l=3 ; m=4

for arbitrary expressions e_1 and e_2?

Non vale in generale ma solo su certe condizioni

$e_1 \equiv \ l = 0 \ , \ l = !l + 1$

$e_2 \equiv \ l = 1$

$$e_1 \parallel e_2 \overset{?}{\approx} e_1 ; e_2 + e_2 ; e_1$$

$$e_1 \parallel e_2 \xrightarrow{l=0} l = !l+1 \parallel l = 1 \xrightarrow{l=1} l = !l+1 \longrightarrow l = 2$$

$$e_1 ; e_2 + e_2 ; e_1 \xrightarrow{l=0} l := l+1 ; l = 1 \xrightarrow{l=1} l = 1$$

Si rompe la simulazione (E)
l'altra resta valida

Example: On Bisimulation

Let

- $e_1 \stackrel{\text{def}}{=} (\mathrm{l} := 1) + (\mathrm{l} := 1; \mathrm{m} := 2)$
- $e_2 \stackrel{\text{def}}{=} \mathrm{l} := 1; \mathrm{m} := 2$

which of the following statements is true?

- $e_1 \sqsubseteq_\Gamma e_2$
- $e_1 \sqsupseteq_\Gamma e_2$

se valgono le simulazioni in entrambi i versi non sempre vale la bisimulazione

- $e_1 \approx_\Gamma e_2$.

se vale la bisimulazione sicuramente valgono le simulazioni in entrambi i versi

$e_1 \approx e_2$

$l=1 \downarrow$ 1   2 $\downarrow l=1$

~~skip~~ $m=2$

$\downarrow$ 3

An encoding of nondeterministic choice

Question: Is $e_1 + e_2$ a primitive construct or it can be codified?

An encoding of nondeterministic choice

Question: Is $e_1 + e_2$ a primitive construct or it can be codified?
Let us try to encode nondeterministic choice using parallel composition, locations and the construct for critical regions:

$$
\underbrace{e_1 \uplus e_2}_{\text{macro}} \stackrel{\text{def}}{=} \begin{array}{l} \text{let } \mathrm{m} : \text{ref int} = \text{ref } 0 \text{ in} \\ \quad \big(\text{await } !\mathrm{m} = 0 \text{ protect } \mathrm{m} := 1 \text{ end}; e_1 \\ \qquad\qquad \| \\ \quad\; \text{await } !\mathrm{m} = 0 \text{ protect } \mathrm{m} := 1 \text{ end}; e_2\big) \end{array}
$$

Said in other words: does our implementation of nondeterministic choice satisfy its specification, or... something close to it? Actually:

- $e_1 \uplus e_2 \;\sqsupseteq_\Gamma\; e_1 + e_2$
- $e_1 \uplus e_2 \;\sqsubseteq_\Gamma\; e_1 + e_2$
- $e_1 \uplus e_2 \;\not\approx_\Gamma\; e_1 + e_2$
- $e_1 \uplus e_2 \;\approx_\Gamma\; (\mathit{skip}; e_1) + (\mathit{skip}; e_2)$.

Persistent behaviours (1)

We know that when we write $e_1; e_2$ we have to execute e_1 first, and only when e_1 has been completed we can execute e_2.

However, how can we write in our language a program that repeats subsequently the same program e?

$$e; e; e; e; \ldots$$

Proposal:

$$\begin{aligned} RepSeq(e) \quad \stackrel{\text{def}}{=} \quad & \text{let } S : (\text{unit} \to \text{unit}) \to (\text{unit} \to \text{unit}) \\ & \quad = \big(\text{fn } f : \text{unit} \to \text{unit} \Rightarrow (\text{fn } x : \text{unit} \Rightarrow x; (f\ x))\big) \\ & \quad \text{in fix}.S\ e \end{aligned}$$

Suppose to have a CBN semantics!

Persistent behaviours (2)

What about a program that forks an arbitrary number of threads e?

$$e \parallel e \parallel e \parallel e \parallel \ldots$$

Proposal:

$$\begin{aligned} RepPar(e) \quad \stackrel{\text{def}}{=} \quad & \text{let } P : (\text{unit} \to \text{unit}) \to (\text{unit} \to \text{unit}) \\ & \quad = \big(\text{fn } f : \text{unit} \to \text{unit} \Rightarrow (\text{fn } x : \text{unit} \Rightarrow x \parallel (f\ x))\big) \\ & \text{in fix}.P\ e \end{aligned}$$

Again suppose to have a CBN semantics!

Data race and critical regions (1)

During the execution of the program $\mathit{RepSeq}(\mathrm{l} := !\mathrm{l} + 1)$ the value associated to the location l increases monotonically:

$$\mathrm{l} := !\mathrm{l} + 1; \mathrm{l} := !\mathrm{l} + 1; \mathrm{l} := !\mathrm{l} + 1; \ldots$$

Whereas during the execution of the program $\mathit{RepPar}(\mathrm{l} := !\mathrm{l} + 1)$ the value associated to the location l may increase or decrease.

$$\mathrm{l} := !\mathrm{l} + 1 \parallel \mathrm{l} := !\mathrm{l} + 1 \parallel \mathrm{l} := !\mathrm{l} + 1 \parallel \ldots$$

This is because this program suffers of data races at locations *l*.
Actually:

può anche scegliere di eseguire uno per volta

- $\mathit{RepSeq}(\mathrm{l} := !\mathrm{l} + 1) \sqsubseteq_\Gamma \mathit{RepPar}(\mathrm{l} := !\mathrm{l} + 1)$
- $\mathit{RepSeq}(\mathrm{l} := !\mathrm{l} + 1) \not\sqsupseteq_\Gamma \mathit{RepPar}(\mathrm{l} := !\mathrm{l} + 1)$

Data races and critical regions (2)

Any way to avoid those data races maintaining concurrency?
Proposal:

$$
\begin{aligned}
&\mathit{AwtPar}(e) \stackrel{\text{def}}{=} \\
&\quad \mathsf{let}\ A : (\mathsf{unit} \to \mathsf{unit}) \to (\mathsf{unit} \to \mathsf{unit}) \\
&\quad\ = \big(\mathsf{fn}\ f : \mathsf{unit} \to \mathsf{unit} \Rightarrow (\mathsf{fn}\ x : \mathsf{unit} \Rightarrow \mathsf{await}\ \mathit{true}\ \mathsf{protect}\ x\ \mathsf{end} \parallel (f\ x))\big) \\
&\quad \mathsf{in}\ \mathsf{fix}.A\ e
\end{aligned}
$$

Now, it is possible to prove that

$$\mathit{RepSeq}(l := !l + 1) \approx_{\Gamma} \mathit{AwtPar}(l := !l + 1).$$