Varese, September 19-21 2012

Hoare Logic for Multiprocessing

(Work in progress)

Daniel Pellarini

joint work with

Marina Lenisa

Università degli studi di Udine, Italy

Hoare Logic

- A formal system, introduced in 1969, used to study the correctness of computer programs
- It makes use of **logical rules** (hence the name), both in the form of **axioms** and **inference rules**
- The verification of computer programs is conducted through the use of **Hoare Triples**, i.e. rules of the following form:

$$\{\ \textit{precondition}\ \}\ \textit{command}\ \{\ \textit{postcondition}\ \}$$

meaning: when we find ourselves in a particular situation (explained by the precondition) and we execute a command, we will end in a state defined by the postcondition

Hoare Logic applied to concurrent programs

- Hoare Logic is a very flexible formal system (it can be used to study sequential, recursive, parallel and even distributed programs)
- This flexibility has a price: it's not very well suited for studying parallel programs (especially the ones with shared variables)

There are two reasons for this:

- in order to study the correctness of a parallel program, we must first transform it into an equivalent sequential one via the interleaving semantics
- the **interference freedom** controls are **very heavy**, forcing us to take care of many different possible computation paths

Refinements for Hoare Logic

- In order to overcome some of these limitations, a number of refinements to the classical Hoare Logic have been introduced, one of which is the so-called **Separation Logic**
- Hoare Triples take a slightly different meaning
- The focus is on the local execution of computer programs: instead of looking at the global state of the system, we only look at the portion of memory that a specific piece of program is addressing
- It's still based on the interleaving semantics

What this work is about

This work is a refinement of Hoare Logic in a different direction: we introduce a sort of **true concurrency operational semantics** and a suitable version of Hoare Logic for the verification of computer programs.

This new operational semantics is not based on interleaving, but it captures a multiprocessing execution.

This is achieved by means of a multiple concurrent execution of the largest possible number of disjoint (or non-racy) processes.

Advantages of this new approach

This new way of handling concurrent programs has a couple of interesting advantages compared to the classical approach, making it faster and simpler:

- there is no need to sequentialize the program to verify its correctness
- among all the possible computation paths, we immediately see those that will never be followed, and we can therefore avoid them
- **the interference freedom checks are done at a local level only**

Language syntax ($\mathcal{L}_{par}$)

Definition (Syntax)

$$(\mathcal{L}_{par} \ni) S ::= \mathit{skip} \mid x := t \mid S_1;\ S_2 \mid \mathit{await}\ B\ \mathit{then}\ S\ \mathit{end} \mid$$
$$\mathit{if}\ B\ \mathit{then}\ S_1\ \mathit{else}\ S_2\ \mathit{fi} \mid \mathit{while}\ B\ \mathit{do}\ S_1\ \mathit{od} \mid [S_1 \mid\mid\mid \ldots \mid\mid\mid S_n]$$

where

- programs not containing the $|||$ operator are called *(sequential) components*;
- *await B then S end* is a conditional atomic section;
- the program S in the *conditional atomic section await B then S end* contains neither the $|||$ operator nor *while* subprograms;
- the components $S_1, \ldots, S_n$ in the parallel composition $[S_1 \mid\mid\mid \ldots \mid\mid\mid S_n]$ do not contain the $|||$ operator
- we consider as atomic actions the skip, assignments, conditional atomic sections and evaluations of Boolean guards.

Operational semantics

Definition (Non-racy atomic actions and parallel transition rule)

(i) We say that two atomic actions A_1 and A_2 are pairwise disjoint (or non-racy) if

- A_1 and A_2 are not conditional atomic sections and
- $change(A_i) \cap var(A_j) = \emptyset$ for $i, j \in \{1, 2\}$ and $i \neq j$

(ii) Multiprocessing parallel transition rule:

$$\frac{\{< S_i,\ \sigma > \longrightarrow < S_i',\ \tau_i >\}_{i \in I}}{< [S_1 \ |||\ \ldots\ |||\ S_n],\ \sigma > \longrightarrow < [T_1 \ |||\ \ldots\ |||\ T_n],\ \biguplus_{i=1}^{n} \tau_i >}$$

where $\{S_i\}_{i \in I}$ is a maximal set of components executing disjoint atomic actions, and $T_i = \begin{cases} S_i & \text{if } i \notin I \\ S_i' & \text{if } i \in I\ . \end{cases}$

Operational semantics - continued

In this operational semantics we could slightly modify the parallel transition rule to allow the execution of an arbitrary number of processes and not necessarily the largest possible one.

This could be useful in a setting with a specific number of processors: if k is the number of processors, we can decide to execute up to k processes concurrently, even if this is not the largest possible set of disjoint components.

One could even think about executing non-racy code sections instead of simple atomic actions, e.g. in a setting of **cloud computing**

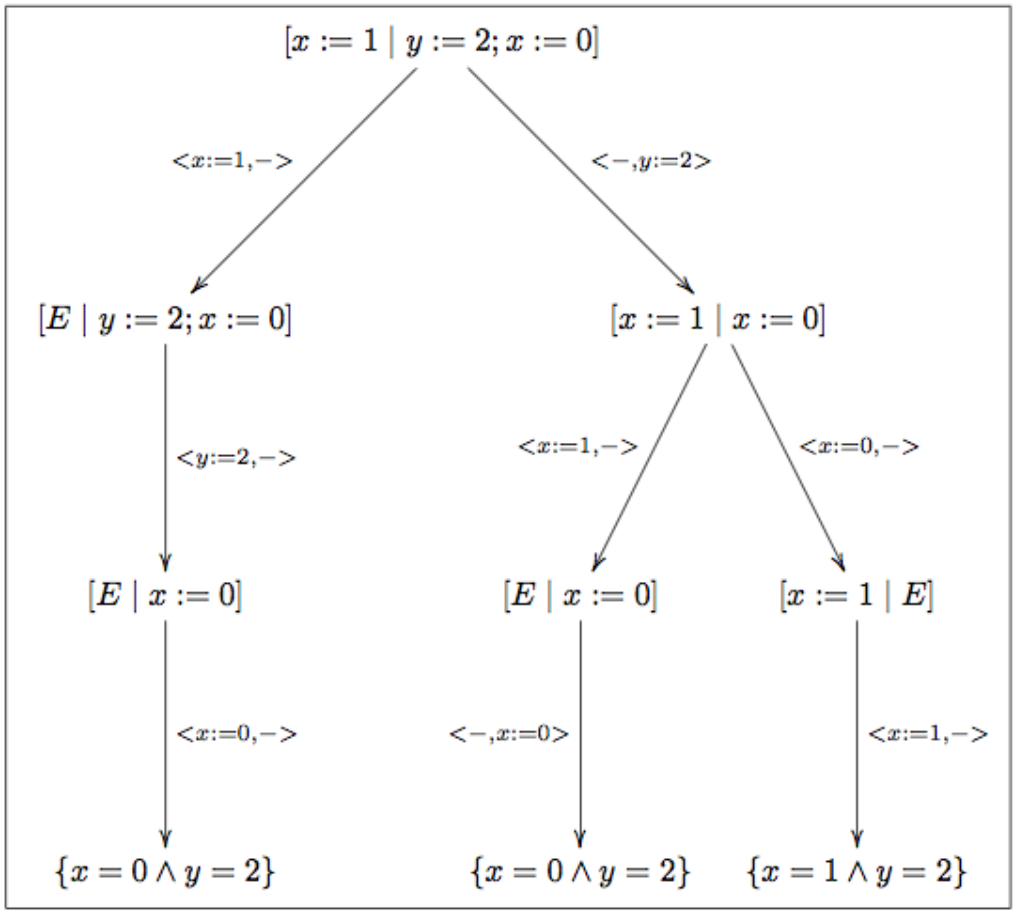
Our multiprocessing semantics is **included in the standard semantics**: we have a smaller number of final states.

Example: interleaving semantics

Let's look at the computations of the simple parallel program

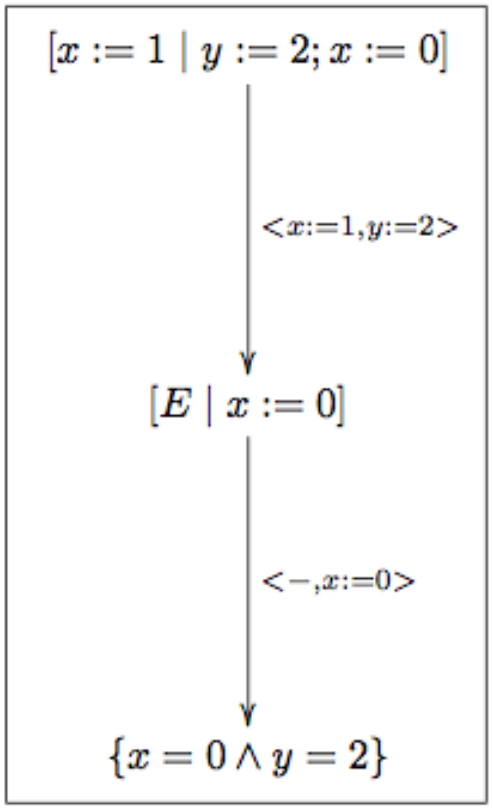
$$< [x := 1 \;|||\; y := 2;\; x := 0],\; \sigma >$$

using the interleaving semantics:



Example: multiprocessing semantics

The same program gives rise to a single computation in the case of multiprocessing semantics:



Verification - some definitions

- A proof outline S^* is a program annotated with assertions
- A proof outline is said to be standard if every subprogram T of S is preceded by exactly one assertion
- We study the correctness of parallel computer programs w.r.t. multiprocessing operational semantics, starting from this notion of proof outline together with the notion of **abstract computation**

The first two definitions are standard in Hoare Logic.

Abstract computation and abstract transition system

Definition (Abstract Transition System and Computation)

(i) The *abstract transition system* consists of rules for deriving judgements $S \xrightarrow{l} S'$, where l is a label representing a list of parallelly-executed atomic actions or the empty action ε, *i.e.*:

$$l \ ::= \ \mathit{skip} \mid x := t \mid B \mid \varepsilon \mid \mathit{await}\ B\ \mathit{then}\ S\ \mathit{end} \mid \langle l_1, \ldots, l_n \rangle \ .$$

(ii) An *abstract computation* is a (finite or infinite) sequence $S \xrightarrow{l_1} S_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} S_n \ldots$

Abstract transition system - rules

Definition

The abstract transition rules are the following:

$$\frac{}{skip \xrightarrow{skip} E} \qquad \frac{}{x := t \xrightarrow{x:=t} E} \qquad \frac{}{S \xrightarrow{\varepsilon} S}$$

$$\frac{}{await\ B\ then\ S\ end \xrightarrow{await\ B\ then\ S\ end} E} \qquad \frac{S_1 \xrightarrow{l_1} S_1'}{S_1; S_2 \xrightarrow{l_1} S_1'; S_2}$$

$$\frac{}{if\ B\ then\ S_1\ else\ S_2\ fi \xrightarrow{B} S_1} \qquad \frac{}{if\ B\ then\ S_1\ else\ S_2\ fi \xrightarrow{\neg B} S_2}$$

$$\frac{}{while\ B\ do\ S\ od \xrightarrow{\neg B} E} \qquad \frac{}{while\ B\ do\ S\ od \xrightarrow{B} S;\ while\ B\ do\ S\ od}$$

$$\frac{\{S_i \xrightarrow{l_i} S_i'\}_{i \in I}}{[S_1 \ ||| \ \ldots \ ||| \ S_n] \xrightarrow{<l_1', \ldots, l_n'>} [S_1' \ ||| \ \ldots \ ||| \ S_n']}$$

where $\{S_i\}_{i \in I}$ is a maximal set of components executing disjoint atomic actions and $l_i' = \begin{cases} l_i & \text{if } i \in I \\ \varepsilon & \text{if } i \notin I \end{cases}$.

Program verification

We study the correctness of parallel programs $[S_1 \,|||\, \ldots \,|||\, S_n]$ in three sequential steps.

Step 1: creation of the proof outlines of the program components. The creation of the single proof outlines is done the usual way: we decorate program components with assertions and invariants, using rules of standard Hoare Logics for partial correctness.

Step 2: creation of the global graph of abstract computations decorated with assertions, and discussion of termination via global termination functions

Step 3: local interference freedom controls

Program verification - Step 2: the graph

Step 2: Given a program $S \equiv [S_1 \ ||| \ \ldots \ ||| \ S_n]$, we build the (finite rooted) graph representing the abstract computations generating from S. Each node n represents a point in the computation of S, and it is labeled by the n-tuple of assertions $\langle p^1_{j_1}, \ldots, p^n_{j_n} \rangle$ appearing in the proof outlines at that point.

- The construction of the graph starts from the root n, which is labeled with the initial assertions
- then, for each created node n' corresponding to $[S'_1 \ ||| \ \ldots \ ||| \ S'_n]$, for each transition $[S'_1 \ ||| \ \ldots \ ||| \ S'_n] \xrightarrow{<l_1,\ldots,l_n>} [S''_1 \ ||| \ \ldots \ ||| \ S''_n]$, a new node n'' corresponding to $[S''_1 \ ||| \ \ldots \ ||| \ S''_n]$ is built, if it does not already exists, and an arc is drawn from n' to n'', labeled by $\langle l_1, \ldots, l_n \rangle$, if the conjunction of all guards appearing in $< l_1, \ldots, l_n >$ and all assertions labeling n' is not false.

Example

Let's study the correctness of the following parallel program:

$$\{\ \textbf{true}\ \}[\ x := 0; \mathit{while}(x \neq 0 \vee y \neq 1)\ \mathit{do}\ x := 1\ \mathit{od}\ |||\ y := 1\ ]\{\ \mathbf{x = 0 \wedge y = 1}\}$$

Notice that the program with interleaving semantics can lead to non termination (even the single component may not terminate). Thus the need of a global termination function for proving the termination of the program with the multiprocessing semantics.

Example - Step 1: proof outlines of the components

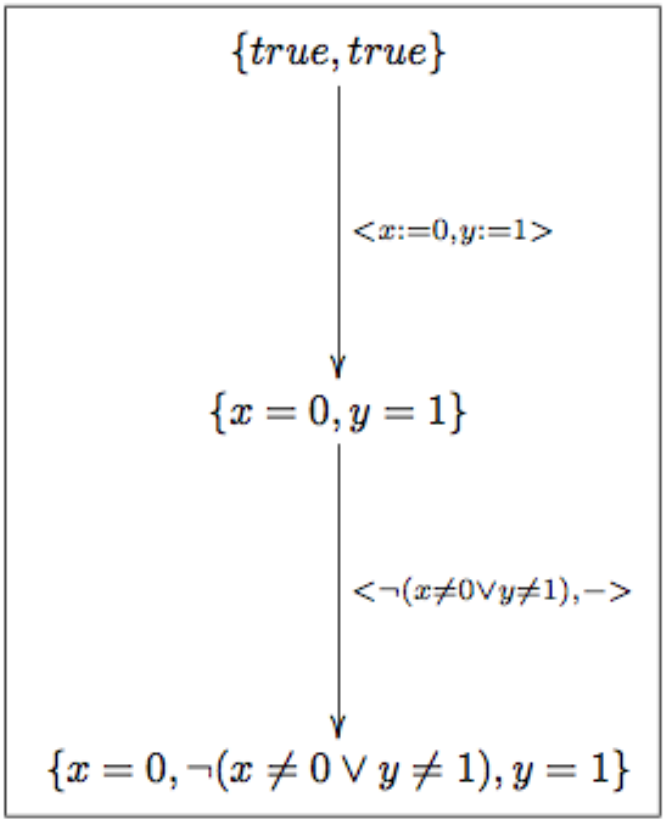
First component

$\{\ true\ \}$
$x := 0$
$\{\ x = 0\ \}\ \{\ inv :\ true\ \}$
while $(x \neq 0 \vee y \neq 1)$ *do* $\{x \neq 0 \wedge y \neq 1\}$
$\quad x := 1$
$\{\ true\ \}$
od
$\{\neg(x \neq 0 \wedge y \neq 1)\}$

Second component

$\{\ true\ \}$
$y := 1$
$\{\ y = 1\ \}$

Example - Step 2: the graph



Program verification - Step 2: the algorithm

$N = \emptyset$
create the root node: $< p_0^1, \ldots, p_0^n >$
add the root node to the set N
while $N \neq \emptyset$ **do**
  take a node $< p_1^1, \ldots, p_1^n >$ corresponding to $[S_1^{'} \mid\mid\mid \ldots \mid\mid\mid S_n^{'}]$ from N
  $N = N \setminus \{< p_0^1, \ldots, p_0^n >\}$
  for all $[S_1^{'} \mid\mid\mid \ldots \mid\mid\mid S_n^{'}] \xrightarrow{<l_1,\ldots,l_n>} [S_1^{''} \mid\mid\mid \ldots \mid\mid\mid S_n^{''}]$ **do**
    if $\bigwedge$ of the guards in the label and $\bigwedge_{i=1}^{n} p_1^i \equiv \mathit{False}$ **then**
      skip
    else
      if the node $< p_2^1, \ldots, p_2^n >$ corresponding to $[S_1^{''} \mid\mid\mid \ldots \mid\mid\mid S_n^{''}] \in N$ **then**
        create edge from $< p_1^1, \ldots, p_1^n >$ to $< p_2^1, \ldots, p_2^n >$ and label it $< l_1, \ldots, l_n >$
      else
        create node $< p_2^1, \ldots, p_2^n >$ and create the edge and label it $< l_1, \ldots, l_n >$
        add the node $< p_2^1, \ldots, p_2^n >$ to N
      end if
    end if
  end for
end while

Program verification - Step 3: local interference freedom controls

Step 3: The non-interference checks can be performed at a **local level**. The final global graph in fact represents all the possible computations generating from the initial parallel program.We can therefore study the interference freedom locally for every branch, by checking that the execution of the current atomic action in a component does not interfere with the assertions appearing in the other components.

Summary

- Refinement of Hoare Logic to be used in a **true concurrency** setting
- Verification of parallel programs by applying a 3-step algorithm based on the notion of abstract computation and abstract transition system
- Local interference freedom controls
- Global termination functions

Future works

- Game semantics of multiprocessing systems
- Coalgebraic semantics
- Other algorithms can be studied using this new technique
- Using this approach in a setting of distributed programming or cloud computing