Software Analysis & Verification **Due: Apr 28, 2020**

Homework 4

Instructor: Fei He
TA: Jianhui Chen, Fengmin Zhu

Chenhao Li (2017011466)

Read the instructions below carefully before you start working on the assignment:

- Please typeset your answers in the attached LaTeX source file, compile it to a PDF, and finally hand the PDF to Tsinghua Web Learning *before the due date*.
- Make sure you fill in your *name* and *Tsinghua ID*, and replace all "`TODO`"s with your solutions.
- Any kind of dishonesty is *strictly prohibited* in the full semester. If you refer to any material that is not provided by us, you *must cite* it.
- Unlike previous assignments, in this one, you will do more reading (and also thinking) than writing.

IMP, Revisited

Before getting start, let's first revisit IMP, *a simple IMPerative programming language*, we have introduced in the lectures. We now present all necessary notations we need in the problems. Some of them may look different from the lectures. We will keep using them in the subsequent assignments and possibly also in the final examination.

Above all, let's specify the syntax:

$$\begin{aligned}
\text{Arithmetic expression } a &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
\text{Boolean expression } b &::= \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
\text{Command/Statement } c &::= \mathsf{skip} \mid x := a \mid c_1; c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \\
&\quad \mid \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}
\end{aligned}$$

where n denotes a constant integer and x denotes a variable. At the top level, an IMP *program* is simply a command.

We put the assignments of variables in a *total map*, i.e. a mapping from variables to integers. Initially, the total map is *empty*, denoted by $\varnothing$, and every variable is mapped to 0, by default. We use the notion $\sigma(x)$ to obtain the integer that variable x maps to. Besides, we can update a total map σ, by letting the variable x map to integer n, yielding a new total map $\sigma[x \mapsto n]$. Trivially, you see $(\sigma[x \mapsto n])(x) = n$ must hold. This total map is usually called a *state*, when we will later talk about program semantics.

Big-step Operational Semantics In the first pass, let's define the semantics of IMP, using big-step operational semantics. For arithmetic expressions, we specify a semantic function $\mathcal{A}[\![a]\!]_\sigma$, which sends an arithmetic expression to its value, i.e. a constant integer, in the state σ:

$$\begin{aligned}
\mathcal{A}[\![n]\!]_\sigma &\triangleq n \\
\mathcal{A}[\![x]\!]_\sigma &\triangleq \sigma(x) \\
\mathcal{A}[\![a_1 + a_2]\!]_\sigma &\triangleq \mathcal{A}[\![a_1]\!]_\sigma + \mathcal{A}[\![a_2]\!]_\sigma \\
\mathcal{A}[\![a_1 - a_2]\!]_\sigma &\triangleq \mathcal{A}[\![a_1]\!]_\sigma - \mathcal{A}[\![a_2]\!]_\sigma \\
\mathcal{A}[\![a_1 \times a_2]\!]_\sigma &\triangleq \mathcal{A}[\![a_1]\!]_\sigma \times \mathcal{A}[\![a_2]\!]_\sigma
\end{aligned}$$

Note that in the right hand side, we also use the standard arithmetic operators $+, -, \times$ for integers.

Remark: By the way, the "$+$" appeared in "$a_1 + a_2$" is a *syntactic symbol* (part of the syntax of an arithmetic expression), whereas the "$+$" appeared in "$\mathcal{A}[\![a_1]\!]_\sigma + \mathcal{A}[\![a_2]\!]_\sigma$" is a *semantic symbol* (the integer addition operation). To be honest, here we make a "harmless abuse of notations", as human beings can understand them.

For boolean expressions, we specify a semantic function $\mathcal{B}[\![b]\!]_\sigma$, which sends a boolean expression to its value, i.e. a constant boolean value (either $\top$ for true, or $\bot$ for false), in the state σ:

$$
\begin{aligned}
\mathcal{B}[\![\mathsf{true}]\!]_\sigma &\triangleq \top \\
\mathcal{B}[\![\mathsf{false}]\!]_\sigma &\triangleq \bot \\
\mathcal{B}[\![a_1 = a_2]\!]_\sigma &\triangleq \mathcal{A}[\![a_1]\!]_\sigma = \mathcal{A}[\![a_2]\!]_\sigma \\
\mathcal{B}[\![a_1 \leq a_2]\!]_\sigma &\triangleq \mathcal{A}[\![a_1]\!]_\sigma \leq \mathcal{A}[\![a_2]\!]_\sigma \\
\mathcal{B}[\![\neg b]\!]_\sigma &\triangleq \neg \mathcal{B}[\![b]\!]_\sigma \\
\mathcal{B}[\![b_1 \wedge b_2]\!]_\sigma &\triangleq \mathcal{B}[\![b_1]\!]_\sigma \wedge \mathcal{B}[\![b_2]\!]_\sigma
\end{aligned}
$$

Note that in the right hand side, we also use the standard comparison operators $=/\leq$ for comparing if two integers are equal/one is less greater than or equal to the other, as well as the standard logical operators $\neg/\wedge$ for computing negation/conjunction.

Finally, for commands (or statements), the evaluation relation is denoted by $\langle \sigma, c \rangle \Downarrow \sigma'$, meaning that executing a command c in a starting state σ results in an ending state σ'. This can also be pronounced "c takes state σ to σ'". Inference rules are as follows:

$$
\begin{aligned}
&(\text{Skip})\ \frac{}{\langle \sigma, \mathsf{skip} \rangle \Downarrow \sigma} \\
&(\text{Ass})\ \frac{\mathcal{A}[\![a]\!]_\sigma = n}{\langle \sigma, x := a \rangle \Downarrow \sigma[x \mapsto n]} \\
&(\text{Seq})\ \frac{\langle \sigma, c_1 \rangle \Downarrow \sigma' \qquad \langle \sigma', c_2 \rangle \Downarrow \sigma''}{\langle \sigma, c_1; c_2 \rangle \Downarrow \sigma''} \\
&(\text{IfTrue})\ \frac{\mathcal{B}[\![b]\!]_\sigma = \top \qquad \langle \sigma, c_1 \rangle \Downarrow \sigma'}{\langle \sigma, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \rangle \Downarrow \sigma'} \\
&(\text{IfFalse})\ \frac{\mathcal{B}[\![b]\!]_\sigma = \bot \qquad \langle \sigma, c_2 \rangle \Downarrow \sigma'}{\langle \sigma, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \rangle \Downarrow \sigma'} \\
&(\text{WhileFalse})\ \frac{\mathcal{B}[\![b]\!]_\sigma = \bot}{\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end} \rangle \Downarrow \sigma} \\
&(\text{WhileTrue})\ \frac{\mathcal{B}[\![b]\!]_\sigma = \top \qquad \langle \sigma, c \rangle \Downarrow \sigma' \qquad \langle \sigma', \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end} \rangle \Downarrow \sigma''}{\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end} \rangle \Downarrow \sigma''}
\end{aligned}
$$

Small-step Operational Semantics Alternatively, we could use small-step operational semantics to repeat the things we have just done. To follow a "pure small-step" style, you may redefine the semantics of arithmetic expressions and boolean expressions in the style of small-step, the evaluation relations may be denoted by $\langle \sigma, a \rangle \to_{\mathcal{A}} a'$ and $\langle \sigma, b \rangle \to_{\mathcal{B}} b'$, respectively. If you attempt this, you will realize that more inference rules are needed than before.

To make our life easier, we could simply reuse the semantic functions $\mathcal{A}[\![a]\!]_\sigma$ and $\mathcal{B}[\![b]\!]_\sigma$. We are now only interested in how to describe the semantics for commands, using small-step operational semantics. The evaluation relation has the form $\langle \sigma, c \rangle \to \langle \sigma', c' \rangle$, meaning that executing a command c, just "one step", from a starting state σ, results in an ending state σ', and the remaining command

we need to execute later is denoted by c'. Inference rules are as follows:

$$\text{(Ass)}\ \frac{\mathcal{A}[\![a]\!]_\sigma = n}{\langle \sigma, x := a\rangle \to \langle \sigma[x \mapsto n], \mathsf{skip}\rangle}$$

$$\text{(SeqStep)}\ \frac{\langle \sigma, c_1\rangle \to \langle \sigma', c_1'\rangle}{\langle \sigma, c_1; c_2\rangle \to \langle \sigma', c_1'; c_2\rangle}$$

$$\text{(SeqFinish)}\ \frac{}{\langle \sigma, \mathsf{skip}; c_2\rangle \to \langle \sigma, c_2\rangle}$$

$$\text{(IfTrue)}\ \frac{\mathcal{B}[\![b]\!]_\sigma = \top}{\langle \sigma, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\rangle \to \langle \sigma, c_1\rangle}$$

$$\text{(IfFalse)}\ \frac{\mathcal{B}[\![b]\!]_\sigma = \bot}{\langle \sigma, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\rangle \to \langle \sigma, c_2\rangle}$$

$$\text{(While)}\ \frac{}{\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \to \langle \sigma, \mathsf{if}\ b\ \mathsf{then}\ (c; \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end})\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}\rangle}$$

Note that when we have reached $\langle \sigma, \mathsf{skip}\rangle$ (for an arbitrary state σ), no rules are applicable and hence we are done. If you wish, we can call this the *normal form*. The so-called "multiple step" relation $\to_*$ is the reflexive and transitive closure of the "one step" relation $\to$.

Now, let's dive into the exercises.

Problem 1: Nondeterministic IMP

IMP's evaluation relation is deterministic. However, nondeterminism is an important part of the definition of many real-world programming languages. For example, in many imperative languages (such as C and its relatives), the order in which function arguments are evaluated is unspecified. The program fragment

```
x = 0;
f(++x, x)
```

might call `f` with arguments `(1, 0)` or `(1, 1)`, depending how the compiler chooses to order things. This can be a little confusing for programmers, but it gives the compiler writer useful freedom. Apart from this, non-determinism is unsurprisingly seen in real-world embedded systems and it can model the unspecified behaviors of complicated network protocols, outside physical environment, etc.

In this exercise, we will extend IMP with a simple nondeterministic command and study how this change affects program equivalence. The new command has the syntax havoc x, where x is a variable. The effect of executing havoc x is to assign an arbitrary integer to the variable x, nondeterministically. For example, after executing the program

$$\mathsf{havoc}\ Y; Z := Y \times 2,$$

the value of Y could be any integer, while the value of Z is twice that of Y (so Z is always even). Note that we are not saying anything about the *probabilities* of the outcomes – just that there are (infinitely) many different outcomes that can possibly happen after executing this nondeterministic code.

In a sense, a variable on which we do havoc roughly corresponds to an uninitialized variable in a low-level language like C. After the havoc, the variable holds a fixed but arbitrary number. Most sources of nondeterminism in language definitions are there precisely because programmers don't care which choice is made (and so it is good to leave it open to the compiler to choose whichever will run faster).

1-1 Based upon the big-step operational semantics of IMP, we need to add *ONE* inference rule for havoc. Please find it out.

Solution

$$\text{(Havoc)}\ \frac{}{\langle \sigma, \mathsf{havoc}\ X \rangle \Downarrow \sigma[X \mapsto n]}$$

■

1-2 Show that the following evaluation relation can hold:

$$\langle \varnothing, \mathsf{skip}; \mathsf{havoc}\ Z \rangle \Downarrow \varnothing[Z \mapsto 42].$$

Proof.

$$\text{(Seq)}\ \frac{\text{(Skip)}\ \dfrac{}{\langle \varnothing, \mathsf{skip} \rangle \Downarrow \varnothing} \qquad \text{(Havoc)}\ \dfrac{}{\langle \varnothing, \mathsf{havoc}\ Z \rangle \Downarrow \varnothing[Z \mapsto 42]}}{\langle \varnothing, \mathsf{skip}; \mathsf{havoc}\ Z \rangle \Downarrow \varnothing[Z \mapsto 42]}$$

□

Problem 2: Don't Loop Forever! Take a Break!

Imperative languages like C and Java often include a break or similar statement for interrupting the execution of loops. In this exercise we consider how to add break to IMP. First, we need to enrich the language of commands with an additional command break.

Next, we need to define the behavior of break. Informally, whenever break is executed in a sequence of commands, it stops the execution of that sequence and signals that the innermost enclosing loop should terminate. If there aren't any enclosing loops, then the whole program simply terminates. The final state should be the same as the one in which the break statement was executed. One important point is what to do when there are multiple loops enclosing a given break. In those cases, break should only terminate the *innermost* loop. Thus, after executing the following program

$$
\begin{array}{l}
X := 0; Y := 1; \\
\textsf{while } \neg(Y = 0) \textsf{ do} \\
\quad \textsf{while true do break end}; \\
\quad X := 1; Y := Y - 1 \\
\textsf{end}
\end{array}
$$

The value of X should be 1, and not 0.

One way of expressing this behavior is to add another parameter, namely *signal*, which is either $\not\triangleright$ ("break") or $\triangleright$ ("continue"), to the (big-step) evaluation relation that specifies whether evaluation of a command executes a break statement. In this way, the evaluation relation now has the form $\langle \sigma, c\rangle \Downarrow \langle \sigma', s\rangle$, meaning that, if c is started in state σ, then it terminates in state σ' and either signals that the innermost surrounding loop (or the whole program) should exit immediately (when $s = \not\triangleright$), or that execution should continue normally (when $s = \triangleright$).

The definition of the "$\langle \sigma, c\rangle \Downarrow \langle \sigma', s\rangle$" relation is very similar to the regular one "$\langle \sigma, c\rangle \Downarrow \sigma'$" – we just need to handle the termination signals appropriately:

- If the command is skip, then the state doesn't change and execution of any enclosing loop can continue normally.
- If the command is break, the state stays unchanged but we signal a $\not\triangleright$.
- If the command is an assignment, then we update the binding for that variable in the state accordingly and signal that execution can continue normally.
- If the command is a if-statement, then the state is updated as in the original semantics, except that we also propagate the signal from the execution of whichever branch was taken.
- If the command is a sequence $c_1; c_2$, we first execute c_1. If this yields a $\not\triangleright$, we skip the execution of c_2 and propagate the $\not\triangleright$ signal to the surrounding context; the resulting state is the same as the one obtained by executing c_1 alone. Otherwise, we execute c_2 on the state obtained after executing c_1, and propagate the signal generated there.
- Finally, for a loop of the form while b do c end, the semantics is almost the same as before. The only difference is that, when the condition b evaluates to true, we execute the body c and check the signal that it raises. If that signal is $\triangleright$, then the execution proceeds as in the original semantics. Otherwise, we stop the execution of the loop, and the resulting state is the same as the one resulting from the execution of the current iteration. In either case, since break only terminates the innermost loop, the entire loop signals $\triangleright$.

2-1 Based on the above description, write a *complete* definition (i.e. inference rules) of the evaluation relation "$\langle \sigma, c\rangle \Downarrow \langle \sigma', s\rangle$". Or, you may read this problem as: "translate the above natural language into formal language".

Solution

$$\text{(Skip)}\ \frac{}{\langle \sigma, \mathsf{skip}\rangle \Downarrow \langle \sigma, \triangleright\rangle}$$

$$\text{(Break)}\ \frac{}{\langle \sigma, \mathsf{break}\rangle \Downarrow \langle \sigma, \not\triangleright\rangle}$$

$$\text{(Ass)}\ \frac{\mathcal{A}[\![a]\!]_\sigma = n}{\langle \sigma, x := a\rangle \Downarrow \langle \sigma[x \mapsto n], \triangleright\rangle}$$

$$\text{(SeqBreak)}\ \frac{\langle \sigma, c_1\rangle \Downarrow \langle \sigma', \not\triangleright\rangle}{\langle \sigma, c_1; c_2\rangle \Downarrow \langle \sigma', \not\triangleright\rangle}$$

$$\text{(SeqCont)}\ \frac{\langle \sigma, c_1\rangle \Downarrow \langle \sigma', \triangleright\rangle \qquad \langle \sigma', c_2\rangle \Downarrow \langle \sigma'', s\rangle}{\langle \sigma, c_1; c_2\rangle \Downarrow \langle \sigma'', s\rangle}$$

$$\text{(IfTrue)}\ \frac{\mathcal{B}[\![b]\!]_\sigma = \top \qquad \langle \sigma, c_1\rangle \Downarrow \langle \sigma', s\rangle}{\langle \sigma, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\rangle \Downarrow \langle \sigma', s\rangle}$$

$$\text{(IfFalse)}\ \frac{\mathcal{B}[\![b]\!]_\sigma = \bot \qquad \langle \sigma, c_2\rangle \Downarrow \langle \sigma', s\rangle}{\langle \sigma, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\rangle \Downarrow \langle \sigma', s\rangle}$$

$$\text{(WhileFalse)}\ \frac{\mathcal{B}[\![b]\!]_\sigma = \bot}{\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \Downarrow \langle \sigma, \triangleright\rangle}$$

$$\text{(WhileTrueBreak)}\ \frac{\mathcal{B}[\![b]\!]_\sigma = \top \qquad \langle \sigma, c\rangle \Downarrow \langle \sigma', \not\triangleright\rangle}{\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \Downarrow \langle \sigma', \triangleright\rangle}$$

$$\text{(WhileTrueCont)}\ \frac{\mathcal{B}[\![b]\!]_\sigma = \top \qquad \langle \sigma, c\rangle \Downarrow \langle \sigma', \triangleright\rangle \qquad \langle \sigma', \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \Downarrow \langle \sigma'', \triangleright\rangle}{\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \Downarrow \langle \sigma'', \triangleright\rangle}$$

■

2-2 Show that for every command c, states σ and σ', and signal s, if

$$\langle \sigma, \mathsf{break}; c\rangle \Downarrow \langle \sigma', s\rangle,$$

then $\sigma = \sigma'$.

Proof.

$$\text{(SeqBreak)}\ \frac{\text{(Break)}\ \dfrac{}{\langle \sigma, \mathsf{break}\rangle \Downarrow \langle \sigma, \not\triangleright\rangle}}{\langle \sigma, \mathsf{break}; c\rangle \Downarrow \langle \sigma, s\rangle}$$

According to the determinacy of big-step operational semantics, $\sigma = \sigma'$. □

2-3 Show that for every command c, states σ and σ', and boolean expression b, if

$$\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \Downarrow \langle \sigma', \triangleright\rangle$$

and $\mathcal{B}[\![\sigma']\!]_b = \top$, then there exists a state σ'' s.t. $\langle \sigma'', c\rangle \Downarrow \langle \sigma', \not\triangleright\rangle$. *Hint: you may need induction in your proof.*

Proof. Since $\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \Downarrow \langle \sigma', \triangleright\rangle$, there must exist a derivation tree with $\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \Downarrow \langle \sigma', \triangleright\rangle$ as the final conclusion. Consider all inference rule with $\langle \sigma, \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{end}\rangle \Downarrow \langle \sigma', \triangleright\rangle$ as its conclusion node:

1. WhileFalse. Then $\sigma = \sigma'$, so $\mathcal{B}[\![b]\!]_\sigma = \top$, which is inconsistent with the premise of WhileFalse.
2. WhileTrueBreak. Then $\langle \sigma, c\rangle \Downarrow \langle \sigma', \not\triangleright\rangle$ must hold, and thus the original proposition is proved.

3. WhileTrueCont. Then there must exist a derivation tree with $\langle \sigma', \textsf{while } b \textsf{ do } c \textsf{ end} \rangle \Downarrow \langle \sigma'', \triangleright \rangle$ as the final conclusion. Since the original derivation tree is finite in its depth, we can apply the same prove technique in finite times, and the final (top level) inferrence is either WhileFalse, which means the premise of the original proposition is violated; or WhileTrueBreak, which means the original proposition is proved.

In conclustion, the original proposition holds in all possible consequences. □

Problem 3: Concurrent IMP

So far, we have done two interesting extensions to IMP: nondeterminism and breaks. Finally, we will see one more – concurrency. At a glance, we will invent a new parallel command

$$\textsf{par } c_1 \textsf{ with } c_2 \textsf{ end}$$

that runs two subcommands c_1 and c_2 in parallel and terminates when both have terminated. To reflect the unpredictability of scheduling, the actions of the subcommands may be interleaved in any order, but they share the same memory and can communicate by reading and writing the same variables.

Unfortunately, the previous elegant big-step operational semantics cannot work in this case. Instead, we use small-step operational semantics and add the following new inference rules:

$$\text{(Par1)}\frac{\langle \sigma, c_1\rangle \to \langle \sigma', c_1'\rangle}{\langle \sigma, \textsf{par } c_1 \textsf{ with } c_2 \textsf{ end}\rangle \to \langle \sigma', \textsf{par } c_1' \textsf{ with } c_2 \textsf{ end}\rangle}$$

$$\text{(Par2)}\frac{\langle \sigma, c_2\rangle \to \langle \sigma', c_2'\rangle}{\langle \sigma, \textsf{par } c_1 \textsf{ with } c_2 \textsf{ end}\rangle \to \langle \sigma', \textsf{par } c_1 \textsf{ with } c_2' \textsf{ end}\rangle}$$

$$\text{(ParDone)}\frac{}{\langle \sigma, \textsf{par skip with skip end}\rangle \to \langle \sigma, \textsf{skip}\rangle}$$

Obviously, we define the binary relation $\to_*$ as the reflexive and transitive closure of $\to$, the one that has included the above three new rules.

Using the above definitions, we can evaluate the program

$$\textsf{par (if true then } X := 1 \textsf{ else } X := 0 \textsf{ fi) with (if true then } Y := 0 \textsf{ else } Y := 1 \textsf{ fi) end}$$

on a starting state σ, which yields a *derivation trace* (until we reach the normal form):

$\langle \sigma, \textsf{par (if true then } X := 1 \textsf{ else } X := 0 \textsf{ fi) with (if true then } Y := 0 \textsf{ else } Y := 1 \textsf{ fi) end}\rangle$
$\to \langle \sigma, \textsf{par } (X := 1) \textsf{ with (if true then } Y := 0 \textsf{ else } Y := 1 \textsf{ fi) end}\rangle$ by (Par1)
$\to \langle \sigma, \textsf{par } (X := 1) \textsf{ with } (Y := 0) \textsf{ end}\rangle$ by (Par2)
$\to \langle \sigma[X \mapsto 1], \textsf{par skip with } (Y := 0) \textsf{ end}\rangle$ by (Par1)
$\to \langle \sigma[X \mapsto 1][Y \mapsto 0], \textsf{par skip with skip end}\rangle$ by (Par2)
$\to \langle \sigma[X \mapsto 1][Y \mapsto 0], \textsf{skip}\rangle$ by (ParDone)

3-1 Give a different derivation trace for the above program. Remember to mention the name of the rule you applied, like we have done above.

Solution

$\langle \sigma, \textsf{par (if true then } X := 1 \textsf{ else } X := 0 \textsf{ fi) with (if true then } Y := 0 \textsf{ else } Y := 1 \textsf{ fi) end}\rangle$
$\to \langle \sigma, \textsf{par (if true then } X := 1 \textsf{ else } X := 0 \textsf{ fi) with } (Y := 0) \textsf{ end}\rangle$ by (Par2)
$\to \langle \sigma, \textsf{par } (X := 1) \textsf{ with } (Y := 0) \textsf{ end}\rangle$ by (Par1)
$\to \langle \sigma[X \mapsto 1], \textsf{par skip with } (Y := 0) \textsf{ end}\rangle$ by (Par1)
$\to \langle \sigma[X \mapsto 1][Y \mapsto 0], \textsf{par skip with skip end}\rangle$ by (Par2)
$\to \langle \sigma[X \mapsto 1][Y \mapsto 0], \textsf{skip}\rangle$ by (ParDone)

■

3-2 Let's now consider a more interesting parallel program involving a loop:

$$P : \textsf{par } (Y := 1) \textsf{ with (while } Y = 0 \textsf{ do } X := X + 1 \textsf{ end) end}$$

Find a property that the above program holds. You are encouraged to propose a property that is as general as possible, but you are *not* asked to do so. However, your property *must* be somehow related to the semantic of P.

Solution If $\langle \sigma, P \rangle$ terminates in $\langle \sigma', \mathsf{skip} \rangle$, then $\sigma'(y) = 1$. ■

3-3 We said that big-step operational semantics cannot work in this parallel case. Why? Briefly explain the reason in *ONE or TWO* sentences.

Solution There is no intermediate excution state in big-step operational semantics, which makes it impossible to interleave the excution of c_1 and c_2 in $\mathsf{par}\ c_1\ \mathsf{with}\ c_2\ \mathsf{end}$. ■