CSCI 5535: Homework Assignment 3

Fall 2013: Due Saturday, October 12, 2013

The purpose of this assignment is to introduce non-determinism in operational semantics and to gain further experience with induction on derivations.

Recall the evaluation guideline from the course syllabus.

_Both your ideas and also the clarity with which they are expressed matter—both in your English prose and your code!_

_We will consider the following criteria in our grading:_

- How well does your submission answer the questions? _For example, a common mistake is to give an example when a question asks for an explanation. An example may be useful in your explanation, but it should not take the place of the explanation._

- How clear is your submission? _If we cannot understand what you are trying to say, then we cannot give you points for it. Try reading your answer aloud to yourself or a friend; this technique is often a great way to identify holes in your reasoning. For code, not every program that "works" deserves full credit. We must be able to read and understand your intent. Make sure you state any preconditions or invariants for your functions._

**Submission Instructions.** Upload to the moodle exactly three files named as follows:

- hw3-YourIdentiKey.pdf with your answers to the written questions. Scanned, clearly legible handwritten write-ups are acceptable. Please no other formats—no .doc or .docx. You may use whatever tool you wish (e.g., \LaTeX, Word, markdown, plain text, pencil+paper) as long as it is legibly converted into a pdf.

- hw3-YourIdentiKey.ml with your OCaml implementation.

- hw3-YourIdentiKey.imp with your IMP test case.

1. **Feedback.** Complete the survey on the linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.
2. **Concurrency.** In this question, we consider an extension of IMP with shared-memory concurrency and locks. The abstract syntax of IMP is extended with three new constructs:

\[
\text{commands } c ::= \ldots \\
| c_1 \parallel c_2 \text{ parallel composition} \\
| \text{lock } x \text{ lock} \\
| \text{unlock } x \text{ unlock}
\]

Parallel composition \( c_1 \parallel c_2 \) says that commands \( c_1 \) and \( c_2 \) can be executed concurrently. A parallel composition \( c_1 \parallel c_2 \) waits for both sub-commands \( c_1 \) and \( c_2 \) to reach a terminal and non-deterministically chooses to continue with either \( c_1 \) or \( c_2 \). You may view \( c_1 \parallel c_2 \) as starting two threads running \( c_1 \) and \( c_2 \), waiting for both to finish, and resulting in non-deterministically choosing the terminal “result” of either \( c_1 \) or \( c_2 \).

To synchronize between concurrently-running commands, we introduce constructs for locking and unlocking. We let all locations (i.e., variables) come equipped with a lock in addition to containing an integer value. The constructs, \text{lock } x \text{ and unlock } x , lock and unlock the lock for location \( x \), respectively.

To model held locks, we change the state \( \sigma \) to now be a pair of a store \( \rho \) and a set of locked locations \( L \).

\[
\begin{align*}
\text{stores} & \quad \rho : \text{Loc} \rightarrow \mathbb{Z} \\
\text{locked sets} & \quad L \subseteq \text{Loc} \\
\text{states} & \quad \sigma ::= \langle \rho, L \rangle
\end{align*}
\]

Note that the state previously consisted only of the store part. The command \text{lock } x adds \( x \) to set of currently held locks \( L \) if it is not already in \( L \) or blocks if \( x \) is in \( L \). The command \text{unlock } x removes \( x \) from the set of currently held locks \( L \). If \( x \) is not in the currently held set of locks \( L \), then \text{unlock } x \) does nothing (i.e., is equivalent to \text{skip}).

These constructs are intended to have the standard semantics in languages with shared-memory concurrency. As an example, the following program

\[
\begin{align*}
\{ & \{ \text{lock } x; y := x; x := y + 1; \text{unlock } x \} \\
| & \{ \text{lock } x; z := x; x := z + 1; \text{unlock } x \} \\
\} & ; \\
\text{print } x
\end{align*}
\]

should print “2.” However, if the \text{lock-unlock} pairs were removed, there are executions (i.e., thread inter-leavings) that result in printing “1.” In particular, if both \( y := x \) and \( z := x \) execute before the writes to \( x \), then the final value of \( x \) will be 1.

We now extend the structural small-step operational semantics. We need to change the judgment form only in that the notion of state \( \sigma \) has changed:

\[
\begin{align*}
\sigma \vdash a & \rightarrow a' \quad \text{arithmetic expression } e \text{ steps to } e' \text{ in state } \sigma \\
\sigma \vdash b & \rightarrow b' \quad \text{boolean expression } b \text{ steps to } b' \text{ in state } \sigma \\
\langle c, \sigma \rangle & \rightarrow \langle c', \sigma' \rangle \quad \text{command } c \text{ in state } \sigma \text{ steps to } c' \text{ in } \sigma'
\end{align*}
\]
(a) Give the structural small-step operational semantics inference rules for the three new concurrency commands.

Hint: A possible solution has 7 new inference rules making use of the judgment form:

\[
\begin{align*}
& c \text{ terminal} \\
\end{align*}
\]

to decide if a command \( c \) is a terminal command.

(b) Extend an interpreter for IMP extended with shared-memory concurrency.

i. Download the Homework 3 code pack from the course web page. The README.txt describes the code pack, like in Homework 2. Modify hw3.ml so that it implements a complete small-step interpreter for “IMP with shared-memory concurrency.” You may build on your code from Homework 2 as a starting point if you wish.

While the extensions from Homework 1 and 2 are part of the extended language, the focus is on the current extension for concurrency.

For your convenience, the imp.ml file has been extended with helper functions to work with the new state type.

To implement non-deterministic choice between two cases, please use a call in OCaml to

\[
\text{Random.bool ()}
\]

The Random module is in OCaml’s standard library for pseudo-random number generation.

The Makefile includes a “make test” target that you should use (at least) to test your work.

ii. Modify the file example.imp so that it contains a “tricky” IMP command (presumably involving concurrency) that can be parsed by our IMP test harness (e.g.,

\[
./imp < \text{example.imp}
\]

should not yield a parse error).

Do not modify any other files. Your submission’s grade will be based on how many of the submitted example.imp files it interprets correctly (in a manner just like the “make test” trials). If your submitted example.imp breaks the greatest number of interpreters (and more than 0!), you will receive extra credit. If there is a tie, all those in the tie will receive the extra credit.

Make sure your code compiles. Code that does not compile will not be graded. If there is some case you cannot get to work, simply comment it out.

The driver has been modified from Homework 2 only to remove the big-step interpreter. As before, the flag --trace that will print a trace of evaluation using your step_com function.

3. Operational Semantics of IMP. This question considers properties about the operational semantics of IMP (without any extensions). As an aid and to ensure we consider the same systems, a listing of the operational semantics of IMP is provided in this section.

Figure 1 presents the big-step operational semantics of IMP commands by defining the judgment

\[
\langle c, \sigma \rangle \downarrow \sigma',
\]
which says that command $c$ in state $\sigma$ evaluates to state $\sigma'$. This judgment relies on evaluation judgments for evaluating arithmetic expressions $a$

$$\langle a, \sigma \rangle \Downarrow n \quad \text{where } n \text{ is an integer}$$

and evaluating boolean expressions $b$

$$\langle b, \sigma \rangle \Downarrow t \quad \text{where } t \text{ is a boolean.}$$

In this figure, we give names to the inference rules for convenience. These rule names are prefixed with E for “evaluation.”

In Figure 2, we consider the structural small-step operational semantics of IMP commands. In particular, we define the following judgment that describes a transition relation:

$$\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle,$$

which says command $c$ in state $\sigma$ steps to command $c'$ and state $\sigma'$. The rule names corresponding to “do” reductions are prefixed with D, while the search rules have names prefixed with S. The judgment $c$ terminal indicates the terminal programs (i.e., those that do not reduce any further).

To express evaluation using small-step operational semantics, we define a judgment that captures some number of steps of the one-step relation:

$$\langle c, \sigma \rangle \rightarrow^* \langle c', \sigma' \rangle$$

The above judgment can be read as command $c$ in state $\sigma$ reduces to $c'$ and $\sigma'$ in some number of steps. This relation is the reflexive-transitive closure of the transition relation $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$ and is defined in Figure 3. The rule names are prefixed with MS for “multi-step.”

(a) **Determinism of Small-Step Evaluation.** Consider our structural small-step operational semantics of IMP (without any extensions). We can show that with these semantics, either we have reached a value or there is a unique next step of evaluation. In other words, if have not reached a value, we can always make progress and evaluation is deterministic.

**Lemma 1** (Deterministic Progress of Arithmetic Expression Evaluation). For all arithmetic expressions $a$ and states $\sigma$, either $a = n$ for some $n$ or there exists a unique $a'$ such that $\sigma \vdash a \rightarrow a'$.

**Proof.** By induction on the structure of $e$. \qed

**Lemma 2** (Deterministic Progress of Boolean Expression Evaluation). For all boolean expressions $b$ and states $\sigma$, either $b = t$ for some $t$ or there exists a unique $b'$ such that $\sigma \vdash b \rightarrow b'$.

**Proof.** By induction on the structure of $b$. \qed

**Theorem 3** (Deterministic Progress of Command Evaluation). For all commands $c$ and states $\sigma$, either $c$ terminal or there exists a unique $c'$ and $\sigma'$ such that $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$. 

4
Figure 1: Big-step operational semantics of IMP.

Figure 2: Small-step operational semantics of IMP.
• For this exercise, prove Theorem 3. You may assume we have proofs for Lemma 1 and Lemma 2.

(b) **Equivalence of Big-Step and Small-Step Evaluation.** Consider our big-step operational semantics and our structural small-step operational semantics of IMP. As a sanity check, we would like to show that the two forms of operational semantics correspond, that is,

\[
\langle c, \sigma \rangle \Downarrow \sigma' \text{ iff } \langle c, \sigma \rangle \rightarrow^* \langle \text{skip}, \sigma' \rangle.
\]

In this exercise, we will build up a proof of this correspondence.

i. We first consider the direction from small-step to big-step semantics (right-to-left in the above). This statement is difficult to prove directly. Instead, we first show that if we have a one-step evaluation in the small-step semantics and from that configuration there is an evaluation in the big-step semantics, then original configuration evaluates to that same result.

**Lemma 4.** If \( \sigma \vdash a \rightarrow a' \) and \( \langle a', \sigma \rangle \Downarrow n \), then \( \langle a, \sigma \rangle \Downarrow n \).

**Lemma 5.** If \( \sigma \vdash b \rightarrow b' \) and \( \langle b', \sigma \rangle \Downarrow t \), then \( \langle b, \sigma \rangle \Downarrow t \).

**Lemma 6.** If \( \mathcal{S} :: \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle \) and \( \mathcal{E} :: \langle c', \sigma' \rangle \Downarrow \sigma'' \), then \( \mathcal{E}' :: \langle c, \sigma \rangle \Downarrow \sigma'' \).

• For this part, prove Lemma 6. You may assume we have proofs for Lemma 4 and Lemma 5.

ii. Now, we can show that if we reach \text{skip} (the terminal program) using the small-step operational semantics, then there is an evaluation using the big-step operational semantics.

**Theorem 7.** If \( \mathcal{M} :: \langle c, \sigma \rangle \rightarrow^* \langle \text{skip}, \sigma' \rangle \), then \( \mathcal{E} :: \langle c, \sigma \rangle \Downarrow \sigma' \).

• For this part, prove Theorem 7.

iii. **Extra Credit.** To complete this correspondence, we can also prove the other direction.

**Theorem 8.** If \( \langle c, \sigma \rangle \Downarrow \sigma' \), then \( \langle c, \sigma \rangle \rightarrow^* \langle \text{skip}, \sigma' \rangle \).

Note that this proof is rather long and requires the introduction of some lemmas.

Please attempt this part only after completing the rest of the homework.