McMaster University
Department of Computing and Software
Dr. W. Kahl

SFWR ENG 3E03
Exercise Sheet 8

Design and Selection of Programming Languages

4 November 2005

Exercise 8.1 — Using Operational Semantics to Prove Incorrectness

The following Hoare triples do not hold.

For each of these Hoare triples, present a derivation in the operational semantics that proves a counterexample to the statement.

(a) $\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$

(b) $\{x \geq -5\}\ z := 5 - x\ \mathbf{;}\ x := z + 2\ \{z \leq 11 \wedge x \geq -3\}$

"Proving a counterexample" for the Hoare triple

$$\{pre\}Prog\{post\}$$

means to derive an assertion

$$\sigma_1(Prog) \Rightarrow \sigma_2$$

involving
– a state σ_1 for which *pre* **holds**, and
– a state σ_2 for which *post* **does not hold**.

Exercise 8.2 — Semantics of Exceptions

We consider a simple imperative programming language with exceptions, with the following **abstract syntax**:

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \mathbf{skip} \\
& | & \mathit{Id}\ \mathbf{:=}\ \mathit{Expr} \\
& | & \mathit{Stmt}\ \mathbf{;}\ \mathit{Stmt} \\
& | & \mathbf{if}\ \mathit{Expr}\ \mathbf{then}\ \mathit{Stmt}\ \mathbf{else}\ \mathit{Stmt} \\
& | & \mathbf{while}\ \mathit{Expr}\ \mathbf{do}\ \mathit{Stmt} \\
& | & \mathbf{throw}\ \mathit{Expr} \\
& | & \mathbf{try}\ \mathit{Stmt}\ \mathbf{catch(}\ \mathit{Id}\ \mathbf{)}\ \mathit{Stmt}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Id} \\
& | & \mathit{Num} \\
& | & \mathit{Bool} \\
& | & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\\
\mathit{Op} & ::= & + \mid - \mid * \mid / \mid \leq \mid \geq \mid < \mid >
\end{array}
$$

(a) Define Haskell datatypes for the abstract syntax of this language.

We still have the following basic semantic domains:

$$
\begin{array}{lll}
\mathit{Val} & = \mathit{Bool} + \mathit{Num} & \text{values} \\
\mathit{Store} & = \mathit{Id} \nrightarrow \mathit{Val} & \text{(simple) stores}
\end{array}
$$

We denote the elements of *Val* by True, False, 0, 1, 2, …

(b) For each of the following, indicate whether it denotes an element of the set *Store*, i.e., a possible *Store* (the notation "$a \mapsto b$" means exactly the pair "(a, b)"):

1. True: ☐ False: ☐ $\{b \mapsto \{\mathsf{True}\}, n \mapsto 0\}$
2. True: ☐ False: ☐ $\{k \mapsto 7, b \mapsto 42, m \mapsto 1001, n \mapsto 1, b \mapsto \mathsf{False}\}$
3. True: ☐ False: ☐ $\{b \mapsto 42, k \mapsto \mathsf{True}\}$
4. True: ☐ False: ☐ $\{k \mapsto 5, b \mapsto \mathsf{True}, s \mapsto \mathbf{skip}\}$
5. True: ☐ False: ☐ $\{\} \times \mathit{Val}$
6. True: ☐ False: ☐ $\{n\} \times \{0\}$
7. True: ☐ False: ☐ $\{n\} \times \{0, 1, 2\}$
8. True: ☐ False: ☐ $\{k, m, n\} \times \{0\}$

From an operational point of view, assuming that the expression e evaluates to the number k, the statement "**throw** e" raises exception k.

We allow **only numbers** as exceptions.

If a statement raising an exception is not enclosed by any "**try** _ **catch**" construct, then this exception immediately leads to program termination with an *uncaught exception.*

If there is an enclosing "**try** _ **catch**" construct, then this is of the shape "**try** _ **catch(** i **)** s_2" for some identifier i and a statement s_2. In that case, execution proceeds immediately to s_2 in an environment where the identifier i is bound to the numerical value of the caught exception.

(c) Write down the *Store* that the statement s_2 executes from when control arrives at s_2 in the following program:

$$k := 100 \ \mathbf{;}\ \mathbf{try}\ q := 42 \ \mathbf{;}\ \mathbf{throw}\ 14 \ \mathbf{;}\ s := q + 1 \ \mathbf{catch(}\ n\ \mathbf{)}\ s_2$$

The statement semantics needs to accommodate the possibility of locally uncaught exceptions. Therefore, the lecture introduced an additional assertion schema for operational semantics:

$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x)$ — execution of statement s starting in state σ_1 can terminate in state σ_2 **rasing exception** x

The lecture also showed that in the Haskell interpreter, statement interpretation **with exceptions** can be implemented via:

$\mathit{interpStmtExc} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ (\mathit{Either}\ \mathit{State1}\ (\mathit{State1}, \mathit{Exc}))$

This function corresponds to the operational semantics in the following way:

$\sigma_1(s) \Rightarrow \sigma_2$	**iff**	$\mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ (\mathit{Left}\ \sigma_2)$
$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x)$	**iff**	$\mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ (\mathit{Right}\ (\sigma_2, x))$
$\neg\exists \sigma_2, x \bullet\ \sigma_1(s) \Rightarrow \sigma_2\ \vee$ $\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x)$	**iff**	$\mathit{interpStmt}\ s\ \sigma_1 = \mathit{Nothing}$

(d) Extend operational semantics of expression evaluation to allow for the possibility that expression evaluation raises exceptions. (In particular, division by zero should be defined to raise exception 24.)

(e) Adapt also the definition of the Haskell interpreter functions accordingly.