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) $\{x \ge -5\} \ z := 5 - x \ \{z \le 11 \land x \ge -3\}$

(b) $\{x \ge -5\} \ z := 5 - x \ ; \ x := z + 2 \ \{z \le 11 \land x \ge -3\}$

"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**:

Stmt ::= $skip$	Expr ::= Id
Id := Expr	Num
Stmt; Stmt	Bool
if Expr then Stmt else Stmt	Expr Op Expr
while Expr do Stmt	
throw Expr	Op ::= + - * / \leq \geq $<$ $>$
try Stmt catch(Id) Stmt	

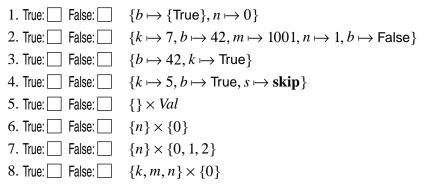
(a) Define Haskell datatypes for the abstract syntax of this language.

We still have the following basic semantic domains:

Val	= Bool + Num	values
Store	$= Id \rightarrow Val$	(simple) stores

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 → b" means exactly the pair "(a, b)"):



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:

$$k := 100$$
; try $q := 42$; throw 14; $s := q + 1$ catch(n) s.

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

 $\sigma_1(s) \stackrel{!}{\Rightarrow} (\sigma_2, x)$ — 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:

interpStmtExc :: Statement \rightarrow State1 \rightarrow Maybe (Either State1 (State1, Exc))

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

$$\begin{aligned} \sigma_{1}(s) \Rightarrow \sigma_{2} & \text{iff} & \text{interpStmt } s \ \sigma_{1} = \text{Just} \ (\text{Left} \ \sigma_{2}) \\ \sigma_{1}(s) \stackrel{!}{\Rightarrow} (\sigma_{2}, x) & \text{iff} & \text{interpStmt } s \ \sigma_{1} = \text{Just} \ (\text{Right} \ (\sigma_{2}, x)) \\ \neg \exists \sigma_{2}, x \bullet \ \sigma_{1}(s) \Rightarrow \sigma_{2} \lor \\ \sigma_{1}(s) \stackrel{!}{\Rightarrow} (\sigma_{2}, x) & \text{iff} & \text{interpStmt } s \ \sigma_{1} = \text{Nothing} \end{aligned}$$

- (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.