Design and Selection of Sequential Programming Languages

14 November 2003

Semantics of Exceptions (mostly from Final 2002)

We consider again the simple imperative programming language with exceptions from Exercise 7.2.

The abstract syntax of this programming language is the following:

\[
\text{Stmt} ::= \text{skip} \quad \text{Expr} ::= \text{Id} \\
| \text{Id} := \text{Expr} \quad | \text{Num} \\
| \text{Stmt} ; \text{Stmt} \quad | \text{Bool} \\
| \text{if} \ \text{Expr} \ \text{then} \ \text{Stmt} \ \text{else} \ \text{Stmt} \quad | \text{Expr} \ \text{Op} \ \text{Expr} \\
| \text{while} \ \text{Expr} \ \text{do} \ \text{Stmt} \\
| \text{throw} \ \text{Expr} \\
| \text{try} \ \text{Stmt} \ \text{catch} (\text{Id}) \ \text{Stmt} \\
\]

We still have the following basic semantic domains:

\[
\text{Val} = \text{Bool} + \text{Num} \quad \text{values} \\
\text{Store} = \text{Id} \rightarrow \text{Val} \quad \text{(simple) stores}
\]

We denote the elements of \text{Val} by True, False, 0, 1, 2, …

(a) For each of the following, indicate whether it denotes an element of the set \text{Store}, i.e., a possible \text{Store} (the notation “a \mapsto b” means exactly the pair “(a, b)”):

1. True: ☐ False: ☐ \{b \mapsto \{\text{True}\}, n \mapsto 0\}
2. True: ☐ False: ☐ \{k \mapsto 7, b \mapsto 42, m \mapsto 1001, n \mapsto 1, b \mapsto \text{False}\}
3. True: ☐ False: ☐ \{b \mapsto 42, k \mapsto \text{True}\}
4. True: ☐ False: ☐ \{k \mapsto 5, b \mapsto \text{True}, s \mapsto \text{skip}\}
5. True: ☐ False: ☐ \{\} \times \text{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.

(b) 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_2
```

The statement semantics needs to accommodate the possibility of locally uncaught exceptions. Therefore, it produces a state transition function that returns either just a Store or a Store together with an exception number, i.e., statement semantics $\llbracket\_\rrbracket_s$ has the following type (the state transition function may be partial to accommodate non-termination):

$\llbracket\_\rrbracket_s : Stmt \rightarrow (Store \rightarrow (Store + (Store \times Num)))$

(c) Define $\llbracket\text{skip}\rrbracket_s$.

(d) Define $\llbracket s_1 ; s_2 \rrbracket_s$ for arbitrary statements $s_1, s_2 : Stmt$.

(e) Define $\llbracket\text{try } s_1 \text{ catch( } i \text{ ) } s_2\rrbracket_s$ for arbitrary statements $s_1, s_2 : Stmt$ and an arbitrary identifier $i : Id$.

Since we already have exceptions in our language, we want primitive operations to raise exceptions if they cannot execute properly, and never produce ⊥.

(f) Propose and explain a type for expression semantics $\llbracket\_\rrbracket_E$.

(g) Define $\llbracket\text{throw } e\rrbracket_s$ for an arbitrary expression $e : Expr$.

(h) Define $\llbracket v := e \rrbracket_s$ for an arbitrary identifier $v : Id$ and an arbitrary expression $e : Expr$.

(i) Define $\llbracket\text{if } b \text{ then } s_1 \text{ else } s_2\rrbracket_s$ for an arbitrary expression $b : Expr$ and arbitrary statements $s_1, s_2 : Stmt$. 