Graduate Programming Languages
Homework Assignment 2

Wes Weimer

Exercise 1: Bookkeeping. Indicate in a sentence or two how much time you spent on this homework, how difficult you found it subjectively, and what you found to be the hardest part. Any non-empty answer will receive full credit.

In addition, give two comments about Pieter Hooimeijer's guest lecture (Machine Learning 1) and two comments about Ray Buse's guest lecture (Machine Learning 2). Any non-empty answer will receive full credit.

Email me your written answers to this homework in a single PDF file named after your UVA ID (e.g., `mst3k.pdf`). You will also be emailing `mst3k-example-imp-command` and `mst3k-hw2.ml`.

> I've just sucked one year of your life away. I might one day go as high as five, but I really don't know what that would do to you. So, let's just start with what we have. What did this do to you? Tell me. And remember, this is for posterity, so be honest — how do you feel? – **Count Rugen, *The Princess Bride***

Exercise 2: Mathematical Induction. Find the flaw in the following inductive proof that "All flowers smell the same". Please indicate exactly which sentences are wrong in the proof. Bringing me a counterexample does not constitute an acceptable solution.

Proof: Let F be the set of all flowers and let $\mathsf{smells}(f)$ be the smell of the flower $f \in F$. (The range of smells is not so important, but we'll assume that it admits equality.) We'll also assume that F is countable. Let the property $P(n)$ mean that all subsets of F of size at most n contain flowers that smell the same.

$$P(n) \stackrel{def}{=} \forall X \in \mathcal{P}(F).\ |X| \leq n \Rightarrow (\forall f, f' \in X.\ \mathsf{smells}(f) = \mathsf{smells}(f'))$$

(the notation $|X|$ denotes the number of elements of X)

One way to formulate the statement to prove is $\forall n \geq 1.P(n)$. We'll prove this by induction on n, as follows:

Base: $n = 1$. Obviously all singleton sets of flowers contain flowers that smell the same (by the definition of $P(n)$).

Induction step:. Let n be arbitrary and assume that all subsets of F of size at most n contain flowers that smell the same. We will prove that the same thing holds for all subsets of size at most $n+1$. Pick an arbitrary set X such that $|X| = n+1$. Pick two distinct flowers $f, f' \in X$ and let's show that $\mathsf{smells}(f) = \mathsf{smells}(f')$. Let $Y = X - \{ f \}$ and $Y' = X - \{ f' \}$. Obviously Y and Y' are sets of size at most n so the induction hypothesis holds for both of them. Pick any arbitrary $x \in Y \cap Y'$. Obviously, $x \neq f$ and $x \neq f'$. We have that $\mathsf{smells}(f') = \mathsf{smells}(x)$ (from the induction hypothesis on Y) and $\mathsf{smells}(f) = \mathsf{smells}(x)$ (from the induction hypothesis on Y'). Hence $\mathsf{smells}(f) = \mathsf{smells}(f')$, which proves the inductive step, and the theorem.

(One indication that the proof might be wrong is the large number of occurrences of the word "obviously" `:-)`)

Exercise 3: Induction. Prove by induction the following statement about the operational semantics:

For any BExp b and any initial state σ such that $\sigma(x)$ is even, if

$$\langle \texttt{while}\ b\ \texttt{do}\ \ x := x + 2, \sigma \rangle \Downarrow \sigma'$$

then $\sigma'(x)$ is even. Make sure you state what you induct on, what the base case is and what the inductive cases are. Show representative cases among the latter. Do not do a proof by mathematical induction!

Exercise 4: Language Features. We extend IMP with a notion of integer-valued *exceptions* (or *run-time errors*), as in Java, ML or C#. We introduce a new type T to represent command terminations, which can either be normal or exceptional (with an exception value $n \in \mathbb{Z}$):

$$\begin{array}{rcll} T & ::= & \sigma & \text{“normal termination”} \\ & | & \sigma\ \mathsf{exc}\ n & \text{“exceptional termination”} \end{array}$$

We use t to range over possible terminations T. We then redefine our operational semantics judgment:

$$\langle c, \sigma \rangle \Downarrow T$$

The interpretation of

$$\langle c, \sigma \rangle \Downarrow \sigma'\ \mathsf{exc}\ n$$

is that command c terminated abruptly by throwing an exception with value $n \in \mathbb{Z}$ at a point in c's execution when the state was σ'. We only model

one type of exception, but every exception has an integer "argument" n (or "payload" or "value") that is set when the exception is thrown and available when the exception is caught.

Note that our previous command rules must be updated to account for exceptions, as in:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \ \mathsf{exc}\ n}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma' \ \mathsf{exc}\ n}\ \mathsf{seq1} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle c_1; c_2, \sigma\rangle \Downarrow t}\ \mathsf{seq2}$$

We also introduce three additional commands:

$$\begin{array}{l}\mathsf{throw}\ e\\ \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2\\ \mathsf{after}\ c_1\ \mathsf{finally}\ c_2\end{array}$$

- The throw e command raises an exception with argument e.
- The try command executes c_1. If c_1 terminates normally (i.e., without an uncaught exception), the try command also terminates normally. If c_1 raises an exception with value e, the variable $x \in L$ is assigned the value e and then c_2 is executed.
- The finally command executes c_1. If c_1 terminates normally, the finally command terminates by executing c_2. If instead c_1 raises an exception with value e_1, then c_2 is executed:
  - If c_2 terminates normally, the finally command terminates by throwing an exception with value e_1. (That is, the original exception e_1 is re-thrown at the end of the finally block, as in Java.)
  - If c_2 throws an exception with value e_2, the finally command terminates by throwing an exception with value e_2. (That is, the new exception e_2 overrides the original exception e_1, also as in Java.)

These constructs are intended to have the standard exception semantics from languages like Java, C# or OCaml — except that the catch block merely assigns to x, it does not bind it to a local scope. So unlike Java, our catch does not behave like a let. We thus expect:

```
x := 0 ;
{ try
    if x <= 5 then throw 33 else throw 55
  catch x
    print x } ;
while true do {
```

```
  x := x - 15 ;
  print x ;
  if x <= 0 then throw (x*2) else skip
}
```

to output "`33 18 3 -12`" and then terminate with an uncaught exception with value `-24`.

- Give the large-step operational semantics inference rules (using our new judgment) for the three new commands presented here. You should present six (6) new rules total.

- Argue for or against the claim that it would be more natural to describe "IMP with exceptions" using small-step contextual semantics. You may use "simpler" or "more elegant" instead of "more natural" if you prefer. Do not exceed two paragraphs (one should be sufficient). Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter.

- Download the Homework 2 code pack from the course web page. Modify `hw2.ml` so that it implements a complete interpreter for "IMP with exceptions (and `print`)". You may build on your code from Homework 1 (although the `let` command is not part of this assignment). Using OCaml's exception mechanism to implement IMP exceptions is actually slightly harder than doing it "naturally", so I recommend that you just implement the opsem rules. The `Makefile` includes a "`make test`" target that you should use (at least) to test your work.

  Hint: to check if a termination `term` is an exception, use syntax like

  ```
  begin match term with
  | Normal -> do_something
  | Exceptional(n) -> do_something_else using n
  end
  ```

- Modify the file `example-imp-command` so that it contains a "tricky" IMP command (presumably involving exceptions) that can be parsed by our IMP test harness (e.g., "`imp < example-imp-command`" should not yield a parse error).

- Rename `hw2.ml` to `uva_email_id-hw2.ml` and rename `example-imp-command` to `uva_email_id-example-imp-command` and email them to me. Do not modify any other files. Your submission's grade will be based on

how many of the submitted `example-imp-command`s it interprets correctly (in a manner just like the "`make test`" trials). If your submitted `example-imp-command` breaks the greatest number of interpreters (and more than 0!), you will receive extra credit. If there is a tie all tiers will receive the extra credit.