CSE 505, Fall 2003, Assignment 1, Sample Solution

(0) See the file `trees.ml`, which implements both the integer and polymorpic trees, using separate names for the functions and type constructors to avoid conflicts and confusion.

(1a) A heap is an OCaml function from strings to integers. The empty heap is function that returns 0 for any input. To look up a variable, we apply the OCaml function to the appropriate string. To set a variable, we create a new function that returns the new value when given the string for the variable. Otherwise the created function calls the function implementing the "old" heap.

(1b) The heap in `interp1.ml` (and `interp2.ml`) does this, but also handles storing constants or statements in variables. A solution to strictly this problem would look like:

```
let mt_heap              = []
let rec get_var heap str =
  match heap with
    [] -> 0
  | (s,v)::tl -> if s=str then v else get_var tl str
let set_var heap str v = (str,v)::heap
```

Note that the only differences between this and the version in `interp?.ml` are the absense of the `heap_val` type and that the empty heap maps variables to `0` rather than `I 0`. Also, it was not necessary to provide both an isolated implementation of the heap for this question and a separate version for question (3b); combining them is perfectly acceptable.

(1c) The interpreter would "forget" any changes to the heap made by a statement nested within the left side of a sequence statement. For example, `ans := 1; skip` would produce a heap where `ans` held 0.

(2a) The following two rules implement the "base case" and the "recursive case" of ntimes. Note the structural similarity to the rules for if.

$$\frac{H\ ;\ e\ \Downarrow\ c \qquad c \leq 0}{H\ ;\ \mathsf{ntimes}\ e\ (s)\ \rightarrow\ H\ ;\ \mathsf{skip}} \qquad \frac{H\ ;\ e\ \Downarrow\ c \qquad c > 0}{H\ ;\ \mathsf{ntimes}\ e\ (s)\ \rightarrow\ H\ ;\ s; \mathsf{ntimes}\ (c-1)\ (s)}$$

(2b) The following rule translates an ntimes statement directly into a while statement, using a fresh variable to control the loop.

$$\frac{x \quad \text{fresh}}{H\ ;\ \mathsf{ntimes}\ e\ (s)\ \rightarrow\ H\ ;\ x := e; \mathsf{while}\ x\ (x := x - 1; s)}$$

(2c.i) False. Consider $H = \cdot, x \mapsto 0$, $s = (\mathsf{while}\ x\ \mathsf{skip}); x := 1$, and $e = 2$ — here s has the property that, for the heap H, it terminates when run once, but modifies the heap to ensure that it will *not* terminate if run a second time on the new heap.

(2c.ii) False. Consider $H = \cdot$, $s = \mathsf{while}\ 1\ \mathsf{skip}$, and $e = 0$ — by letting $e = 0$, the statement $\mathsf{ntimes}\ e\ (s)$ trivially terminates, even if s alone would not.

(2c.iii) True.

Informally, this is because the statement s terminates for *all* heaps H. Therefore nothing one iteration of s does to modify the heap can prevent a future iteration from terminating. Since the number of iterations is fixed, and each iteration eventually terminates, the entire construct must also terminate. The proof is as follows:

From the theorems in lecture, we know that for all H and e, $H\ ;\ e\ \Downarrow\ c$ for some c. The proof is by induction on that constant c.

The base case is trivial: If $c \leq 0$, then for any s, $H; \mathsf{ntimes}\ e\ (s)$ terminates after one step (using the rules in 2a).

If $c > 0$, then $\mathsf{ntimes}\ e\ (s)$ becomes $s; \mathsf{ntimes}\ (c-1)\ (s)$. By assumption, $H; s$ terminates, so using the semantics of sequence statements, we know $H; s; \mathsf{ntimes}\ (c-1)\ (s) \rightarrow^* H'; \mathsf{ntimes}\ (c-1)\ (s)$ for some H'. The induction hypothesis applies to H' and $\mathsf{ntimes}\ (c-1)\ (s)$, so $H'; \mathsf{ntimes}\ (c-1)\ (s)$ terminates. So by the definition of $\rightarrow^*$, $H; \mathsf{ntimes}\ e\ (s)$ terminates.

(2c.iv) False. Consider $H = \cdot$, $s = \mathsf{while}\ 1\ \mathsf{skip}$, and $e = 0$ — this is the same counterexample we used for (2c.ii), and is exploiting the same hole of $e = 0$.

(2d) See the file `interp1.ml`, lines 39–43.

(3a) Following the hint, these rules extend the semantics of heaps to allow variables to map to either constants or expressions (first line of rules); extend the lookup operation to handle these two cases (second line); and extend the operational semantics to handle assignment of code pointers (third line, first rule) and execuation of proper and "improper" code pointers.

$$H ::= \cdot \mid H, x \mapsto c \mid H, x \mapsto s \qquad H(x) = \begin{cases} c & \text{if} \quad H = H, x \mapsto c \\ s & \text{if} \quad H = H, x \mapsto s \\ H'(x) & \text{if} \quad H = H', y \mapsto c' \\ 0 & \text{if} \quad H = \cdot \end{cases}$$

$$\frac{H(x) = c}{H\ ;\ x\ \Downarrow\ c} \qquad \frac{H(x) = s}{H\ ;\ x\ \Downarrow\ 0}$$

$$\frac{}{H\ ;\ x := (s)\ \rightarrow\ H, x \mapsto s\ ;\ \mathsf{skip}} \qquad \frac{H(x) = s}{H\ ;\ \mathsf{run}\ x\ \rightarrow\ H\ ;\ s} \qquad \frac{H(x) = c}{H\ ;\ \mathsf{run}\ x\ \rightarrow\ H\ ;\ \mathsf{skip}}$$

(3b) See the file `interp1.ml`, most particularly lines 44–48, but this change also affects the heap, requiring changes there, and in the places where the heap is read (lines 19–22 and 52, at least), and written (line 29, at least).

(3c) $x := (ans := 42); \mathsf{run}\ x$

(4a) See the file `interp2.ml`, particularly lines 23 and 48.

(4b) See the file `interp2.ml`, lines 58–97. The "heavy lifting" is done by the recursive `get_vars_stmt` function, which walks the abstract syntax tree building four lists (packaged into a single data type) with the variables appearing in expressions, assignments, statement assignements, and run statements. Note that the `@` operator is for list concatenation, so the `combine_vars` function is for `vars` concatention. `prevent_error` then uses the built-in list iteration functions to walk the lists looking for conflicts.

(4c) $x := 3; y := x; x := \mathsf{skip}; \mathsf{run}\ x$. Another interesting one is $x := 1; x := 0; \mathsf{run}\ x$.

(4d.i) Yes. Because `prevent_error` is a conservative analysis to prevent runtime errors (a fact which we asserted in question (4c), although we have not formally proven), we know that no program for which `prevent_error` s is true will ever raise a runtime error. Since the only situation in which the interpreters behave differently are the runtime error cases, no program for which `prevent_error` is true can have different behavior under the two interpreters.

(4d.ii) No. The language is Turing complete, even without code pointers. If s is an arbitrary code-pointer free program, then $s; x := \mathsf{skip}; y := x + 1$ generates an error if and only if s terminates, which is undecidable.