Solutions

1. Environment Model [27 pts] (parts a–d)

   Consider the following code:

```
let val y = (ref "hello", "goodbye")
    val z = ((#1 y) := (#2 y); 0)
    fun f(x: int) = #2 y
    fun g(y: int) = ref f
in
    g(5)
end
```

   (a) [2 pts] What is the type of `f`?

   Answer:

   `int->string`

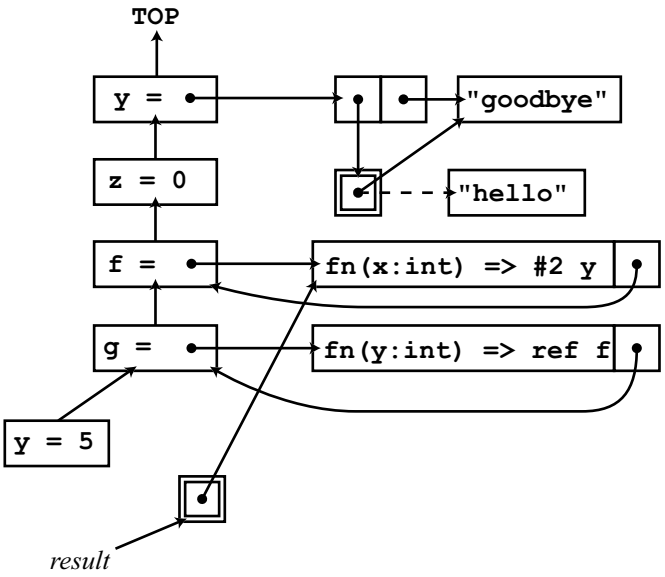
   (b) [2 pts] What is the type of `g`?

   Answer:

   `int->((int->string) ref)`

   (c) [18 pts] Draw the result produced by evaluating this expression *in the environment model*.

   Answer:



   (d) [5 pts] What garbage (other than environment entries) is generated by evaluating this program?

   Answer: The string `hello` and the closure for `g`. Other allocated structures that are still reachable from the result (and therefore not garbage) include the `ref` allocated in `g`, the closure for `f`, the tuple for `y`, and the string `goodbye`.

2. Data abstraction [33 pts] (parts a–d)

   Suppose we want to implement a game of N-by-N tic-tac-toe using a mutable data abstraction for the board. The following is a start at an interface:

```
(* A board is a mutable N-by-N tic-tac-toe board. *)
type board
datatype contents = X | O | Empty
(* A cell is a cell coordinate, from (1,1) up to (N,N) *)
type cell = int * int
(* create(n) creates an n-by-n board with all cells empty. *)
val create: int -> board
(* The number of cells in one row or column of the board.  *)
val boardSize: board -> int
(* The number of non-empty cells. *)
val moves: board -> int
(* The contents of a board cell. *)
val getCell: board*cell -> contents
(* Set the contents of a board cell.
   Requires: that cell is currently empty. *)
val setCell: board*cell*contents -> unit
(* Return whose move it is (always X or O) *)
val whoseMove: board -> contents
```

(a) [5 pts] Classify each of these operations as a creator, observer, or mutator.

Answer: `create` is a creator, `boardSize`, `moves`, `getCell`, and `whoseMove` are observers, and `setCell` is a mutator.

(b) [7 pts] Supply any missing preconditions.

Answer: `create` requires that n is positive (or at least nonnegative). `getCell` and `setCell` require that the `cell` argument is valid, that is, that both coordinates are between 1 and N. There are two ways to make sure `whoseMove` is possible. One is to require in `setCell` that the new contents correspond to the player whose move it is. An alternative is to require in `whoseMove` that the board is one formed by alternating calls to `setCell`.

Consider the following representation:

```
type board = { size: int,
               X's: cell list ref,
               O's: cell list ref }
```

Using this rep, here is how we might implement the function `create` so that it takes only $O(1)$ time in the board size:

```
fun create(n: int) = { size = n, X's = ref nil, O's = ref nil }
```

However, some of the other operations are not so easy to implement.

(c) [15 pts] Give an appropriate representation invariant for this representation. Think about what will be needed to implement all of the functions in the interface above.

Answer: The field `size` must be at least 1. All of the cells in the two lists must have coordinates that are between 1 and `size`, inclusive. No two list elements have the same coordinates. If we don't impose the precondition on `setCell`, we must also require that the lengths of the lists referred to by `X's` and `O's` either are equal or the `X's` list is one longer (if it is O's move).

(d) [6 pts] Suggest a different representation that would permit all of the operations except `create` to be implemented in time $O(1)$ in the board size.

Answer: Here is one way to do it:

```
type board = { cells: contents array array,
               num_moves: int }
```

Note that the parity of the `num_moves` field should be sufficient to decide whose move is next assuming that X always moves first.

3. Recurrences [20 pts] (parts a–b)

The conventional algorithm for multiplying two square matrices of size n takes $O(n^3)$ time. However, there is an asymptotically more efficient algorithm in which the matrix is divided into smaller matrices of size $\frac{n}{2}$ by $\frac{n}{2}$ and 7 matrix multiplications are performed on them. Thus, we arrive at the following recurrence:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 7\,T(n/2) \end{aligned}$$

To simplify analysis, let us consider values of n that are powers of two.

(a) [6 pts] Is the solution to this recurrence $O(n^3)$? Justify your answer briefly.

Answer: Yes, as suggested by the claim above that the algorithm is asymptotically more efficient than an $O(n^3)$ algorithm. To see this, let's try the substitution method. We want to substitute kn^3 for $T(n)$ in the recurrence above:

$$7\,T(n/2) = 7k(n/2)^3 = (7/8)kn^3$$

Because $kn^3 \geq (7/8)kn^3$ (for example, if $k = 1$), and the function kn^3 is monotonic in n, we know that $T(n) \leq kn^3$. The usual inductive argument is used to justify the substitution method: $kn^3 \geq (7/8)kn^3 \geq 7T(n/2) = T(n)$, so $T(n)$ is $O(n^3)$.

We can also verify this "experimentally". Consider the sequence $n = 1, 2, 4, 8, 16, \ldots$. In this case we see $T(2^i) = 7^i$ and $n^3 = 8^i$, so $T(n)$ is clearly bounded above by n^3.

(b) [14 pts] Find the value of c such that the solution to the recurrence is $\Theta(n^c)$.

Answer: Again, we can use the substitution method. We want to compare kn^c with the result of substituting kn^c into the right-hand side of the recurrence:

$$7\,T(n/2) = 7k(n/2)^c = (7/2^c)kn^c$$

So if $1 = 7/2^c$, this is equal to kn^c, and we have obtained both a lower and an upper bound on $T(n)$. Therefore, the solution is $c = \lg 7 \approx 2.8$. In practice, this algorithm is only faster than the usual algorithm for very large matrices.

4. Type checking [20 pts] (parts a–c)

(a) [7 pts] Define a function `f` with type `('a*'b) ref -> ('a ref)*('b ref)`. Remember that this function must be polymorphic.

Answer:

```
fun f(x: ('a*'b) ref) = let val (y,z) = !x in
  (ref y, ref z)
end
```

(b) [3 pts] Give an example of two type expressions that contain unsolved type variables but that cannot be unified.

Answer: Two expressions cannot be unified if there is no way to make them equal by consistently replacing unknowns (type variables in this case) with expressions. For example, the following two type expressions (where T is an unknown type) cannot be unified: T`-> bool`, `int *` T

(c) [10 pts] Consider the following SML function:

```
fun f(x,y,z,w) =
    if z(x) then (x, (y,z)) else (z(x), w)
```

If we let the SML type inference algorithm reconstruct types for this definition, what will be the types inferred for the identifiers `x`, `y`, `z` and `w`?

Answer:

```
x: bool
y: 'a
z: bool->bool
w: 'a * (bool->bool)
```