G6021: Comparative Programming

Exam Practice 3

1. Given the λ-terms: $I = \lambda x.x$, $T = \lambda x.xIII$ and $U = \lambda zy.y(\lambda x.xz)$:

   (a) Write the λ-term TU out in full (without abbreviations, and including ALL brackets).

   Answer:
   $((\lambda x.(((x(\lambda x.x))(\lambda x.x))(\lambda x.x)))(\lambda z.(\lambda y.(y(\lambda x.(xz))))))$

   (b) Draw the β-reduction graph of the λ-term TU.

   Answer: There is only one reduction at each step:

   $$\begin{array}{rcl} TU & \to & (\lambda y.y(\lambda x.xI))II \\ & \to & (I(\lambda x.xI))I \\ & \to & (\lambda x.xI)I \\ & \to & II \\ & \to & I \end{array}$$

   (c) By building a type derivation, find the type of the term U.

   Answer: Let $T = ((A \to B) \to B) \to C$ in the following:

   $$\dfrac{\dfrac{\dfrac{}{z : A, y : T \vdash y : T} \qquad \dfrac{\dfrac{\dfrac{}{z : A, y : T, x : A \to B \vdash x : A \to B} \qquad \dfrac{}{z : A, y : T, x : A \to B \vdash z : A}}{z : A, y : T, x : A \to B \vdash xz : B}}{z : A, y : T \vdash \lambda x.xz : (A \to B) \to B}}{z : A, y : T \vdash y(\lambda x.xz) : C}}{z : A \vdash \lambda y.y(\lambda x.xz) : (((A \to B) \to B) \to C) \to C}}{\vdash \lambda zy.y(\lambda x.xz) : A \to (((A \to B) \to B) \to C) \to C}$$

   (d) For λ-terms X and Y, explain how the pair (X, Y) can be represented as a λ-term. Include in your answer λ-terms to build and to project each component of a pair. Write (T, U) in your chosen encoding.

   Answer: There are several ways to do this, but the following a standard one:
   pair $= \lambda uvx.xuv$,
   fst $= \lambda x.x(\lambda xy.x)$, **snd** $= \lambda x.x(\lambda xy.y)$.
   (T, U) can then be encoded as: **pair**$TU \to^* \lambda x.xTU = \lambda x.x(\lambda x.xIII)(\lambda zy.y(\lambda x.xz))$.
   For "fun", check that **fst**(**pair**AB) $\to^*$ A and **snd**(**pair**AB) $\to^*$ B, for any A, B.

2. (a) Write a function `rev` in Haskell syntax that uses an accumulating parameter to reverse a list. Include the type of your function in your answer.

   Answer:

   ```
   rev :: [a] -> [a] -> [a]
   rev [] acc = acc
   rev (x:s) acc = rev s (x:acc)
   called with rev l []
   ```

(b) Write a function `equal` in Haskell syntax that takes two lists of elements (where each element has a type that is an instance of the `Eq` class) and checks whether they are equal (i.e., returns `True` if they have exactly the same elements in the same order, `False` otherwise). Give the most general (polymorphic) type for `equal`.

Answer:

```
equal :: Eq a => [a] -> [a]  -> Bool

equal []     []  = True
equal (x:s)  []  = False
equal []  (y:p)  = False
equal (x:s)(y:p) = (x == y) && (equal s p)
```

(c) Using `equal` and `rev` write a function `palindrome` that checks whether a list is a palindrome. A list is a palindrome if the list is the same in reverse. The lists [1,0,0,1], [True, False, True] and [0,1,2,3,3,2,1,0] are examples of palindromes.

Answer: Application of ideas developed in previous parts.

```
palindrome :: [a] ->  Bool
palindrome l = equal l (rev l [])
```

(d) Write a set of Prolog clauses that can be used to test if a list is a palindrome. State clearly what your predicates represent.

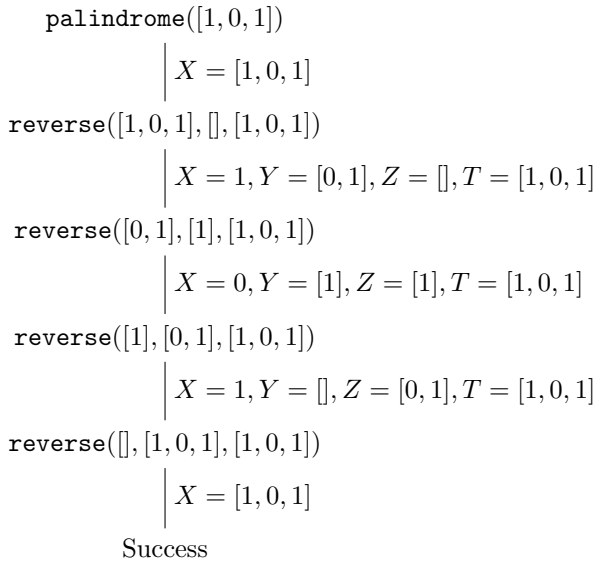
Answer: A number of answers possible, for instance:

```
palindrome(X) :- reverse(X,[],X).

reverse([],X,X).
reverse([X|Y],Z,T) :- reverse(Y,[X|Z],T).
```

(e) Using your answer to part (d), give two example SLD trees to demonstrate your clauses, using the lists: [1,0,1] and [1,0,0].

Answer:



$$\texttt{palindrome}([1,0,0])$$
$$\Big| \; X = [1,0,1]$$
$$\texttt{reverse}([1,0,1],[],[1,0,0])$$
$$\Big| \; X = 1, Y = [0,1], Z = [], T = [1,0,0]$$
$$\texttt{reverse}([0,1],[1],[1,0,0])$$
$$\Big| \; X = 0, Y = [1], Z = [1], T = [1,0,0]$$
$$\texttt{reverse}([1],[0,1],[1,0,0])$$
$$\Big| \; X = 1, Y = [], Z = [0,1], T = [1,0,0]$$
$$\texttt{reverse}([],[1,0,1],[1,0,0])$$
$$\Big|$$
$$\text{Fail}$$

3. (a) Explain the concept of *dynamic lookup*. To what extent is this concept related to *overloading*?

   Answer: Dynamic Lookup - when a message is sent to an object, the method executed is determined by the object implementation. Different objects can respond differently to the same message. The response is not based on the static property of the variable or pointer.
   Related to overloading to some extent, but overloading is a *static* concept: it is the static type information that dictates which code is executed.

   (b) Explain the main problem associated with multiple inheritance, and possible solutions to this problem. Provide examples to support your explanations.

   Answer:
   The example should exhibit *name clashes* (a simple example will suffice).
   Possible solutions:
   - Implicit resolution, defined by language semantics.
   - Explicit resolution, i.e. programmer decides.
   - Disallow multiple inheritance in the language.

   (c) Explain how to write the Haskell function

   ```
   f x = if x==0 then 1 else x*f(x-1)
   ```

   as the fixed point of a functional. Use the syntax of PCF, and include types in your answer.

   Answer:

   $$\mathsf{fix}_{\mathtt{int}\to\mathtt{int}}\Big(\lambda f^{\mathtt{int}\to\mathtt{int}}.\lambda x^{\mathtt{int}}.\mathsf{cond}_{\mathtt{int}}(\mathsf{iszero}\ x)\ 1\big(\mathsf{mult}\ x(f(\mathsf{pred}\ x))\big)\Big)$$

   (d) Define CPS, and explain how to convert the Haskell function:

   ```
   f x = if x==0 then 1 else x*f(x-1)
   ```

   so that it can compiled into a loop (i.e., without recursion).

   Answer:
   The idea of CPS is that every function takes an extra argument: a continuation.
   A continuation is a function which consumes the result of a function, and

produces the final answer. Thus, a continuation represents the remainder of the current computation.
CPS form:

```
factcps n k = if n==0 then k 1
              else factcps (n-1) (\r -> k (n*r))
factcps 4 (\x -> x)
```

We can simplify the continuation:

```
factacc n acc = if n==0 then acc
                else factacc (n-1) (n*acc)
factacc 4 1
```

which can then be compiled as a loop (tail recursive).