Foundations of Computer Science

Supervision 2 – Solutions

4. More on lists

4.1. Conceptual questions

1. The functions zip and unzip seem like they are each other's opposites. Mathematically, two functions $f : A \to B$ and $g : B \to A$ are inverses of each other if for all arguments $a \in A$, $g(f(a)) = a$, and if for all arguments $b \in B$, $f(g(b)) = b$. If only the first condition holds, we call g the *left inverse* of f, and if only the second condition holds, we call g the *right inverse* of f. Is unzip the inverse, right inverse or left inverse of zip? Justify your answer.

   > The two functions would be inverses of each other if their composition was the identity in both directions, for any input. However, we can exhibit counterexamples for this: if two lists being zipped have different lengths, the "surplus" elements are dropped, so we would not get back the two full lists after unzipping:
   >
   > ```
   > unzip (zip ([1;2;3;4], [5;6])) = unzip [(1,5); (2,6)]
   >          = ([1;2],     [5;6])
   > ```
   >
   > Hence we know that unzip cannot be the left inverse of zip, and hence it cannot be its full inverse. However, as unzipping cannot drop any elements (the resulting two lists must have the same length), we do have `zip (unzip l) = l` for all inputs, e.g.
   >
   > ```
   > zip (unzip [(1,5); (2,6); (3,7)]) = zip ([1;2;3], [5;6;7])
   >          = [(1,5); (2,6); (3,7)]
   > ```
   >
   > This can be formally proved using *structural induction* on lists (see later), but should be clear by looking at the definition of the two functions. Therefore we can conclude that unzip is the right inverse of zip.

2. We know nothing about the functions f and g other than their polymorphic types:

   ```
   > val f : 'a * 'b -> 'b * 'a = <fun>
   > val g : 'a -> 'a list = <fun>
   ```

   Suppose that `f (1, true)` and `g 0` are evaluated and return their results. State, with reasons, what you think the resulting *values* will be, and how the functions can be defined. Can any of the definitions be changed if "return their results" wasn't a requirement?

   > The crux of this exercise is that polymorphism provides flexibility for the user but serious constraints for the implementer. If the user should be able to call a function on an input of

any type (e.g. the reverse function on a list of any type), the writer of the function cannot assume anything about the input, i.e. they cannot call any non-polymorphic function on those arguments. This severly limits the ways in which a polymorphic function can be implemented, but this is a good and desired thing – the stricter our type system, the more bugs it can catch and avoid.

In this example, the function f has a tuple argument and a tuple result, with the polymorphic types flipped around. As we cannot assume anything about the type of the arguments, we cannot modify them in any way – the only thing we can do is swap them around in a pair.

```
let f (a,b) = (b,a)
```

This is similar to how the only possible function of the type 'a -> 'a is the identity function: it cannot examine or modify an argument of polymorphic type, so the only thing it can do is return it unchanged.

Polymorphism is not just (or necessarily) about constraining the *number* of ways a function can be implemented, but the *way* in which an implementation can be given. For example, the second function g can have an infinite number of implementations, but they are all quite boring: we can either just return the empty list (which would actually have the more general type 'a -> 'b list), or the singleton list containing the input once, or the two-element containing two copies of the input, and so on. Any return value will simply be a list of repetitions of the input – again, because we don't know anything about the type of the argument, so we cannot modify it in any way.

There is actually another way to implement g, but even though the definition typechecks, it would not return anything: an infinite list of repetitions of the input, defined as a recursive function

```
let rec g x = x :: g x
```

Just as loop in the previous exercise sheet, the typechecker is more "lenient" if a function doesn't terminate, as there is no harm that can be done even if the return value of the function is absurd – it never returns anyway.

4.2. Exercises

3. a) Use the member function on Page 32 to implement the function inter that calculates the *intersection* of two OCaml lists – that is, inter xs ys returns the list of elements that occur both in xs and ys.

   We recurse through one list and use the membership test to check whether the element is in the other list as well.

   ```
   let rec inter xs ys = match xs with
   ```

```
  | []    -> []
  | x::xs -> if member x ys then x::inter xs ys
                            else    inter xs ys
```

b) Similarly, code a function to implement set union. It should avoid introducing repetitions, for example the union of the lists [4;7;1] and [6;4;7] should be [1;6;4;7] (though the order should not matter).

> We can write a function that resembles inter from Slide 404, except the base case returns the nonempty list, and if the head added to the union if it does not appear in the second list:
>
> ```
> let rec union xs ys = match xs with
>   | []    -> ys
>   | x::xs -> if member x ys then    union xs ys
>                             else x::union xs ys
> ```
>
> Returning the second argument if the first list is empty somewhat resembles how we return the accumulator in the base case of tail-recursive functions. In fact, we can write union tail-recursively, but treating the second list as the accumulator – we simply move the new elements from the first list to the second:
>
> ```
> let rec union xs ys = match xs with
>   | []    -> ys
>   | x::xs -> union xs (if member x ys then ys else x::ys)
> ```

4. Code a function that takes a list of integers and returns two lists, the first consisting of all nonnegative numbers found in the input and the second consisting of all the negative numbers. How would you adapt this function so it can be used to implement a sorting algorithm?

> The simplest solution is similar to unzip, where we want to pattern-match on the result of the recursive call. This can be done either with case-expressions (upcoming), or local bindings. We can avoid the binding by using tail-recursion with two accumulators, but then we would have the resulting lists in reverse order.
>
> ```
> let rec split = function
>   | []    -> ([], [])
>   | x::xs -> let (nonneg, neg) = split xs
>              in if x < 0 then (   nonneg, x::neg)
>                          else (x::nonneg,    neg)
> ```
>
> The implementation is very similar to the partition step of quicksort, except it “hardcodes”

> the pivot to 0. To make it more general, we should take the pivot as an argument and perform the comparison on it.

4.3. Optional questions

5. How does this version of `zip` differ from the one in the course?

```
let rec zip = function
  | (x::xs, y::ys) -> (x,y) :: zip (xs,ys)
  | ([], [])       -> []
```

> Compiling this function gives a "Matches not exhaustive" warning, which explains the problem with this definition. The two patterns we declare are either for two nonempty lists, or two empty lists. The function would work fine if the two lists were of the same length – every recursive call would happen on the two tails of the lists (which also have equal length) so eventually the lists will become empty simultaneously. However, if one of the lists is shorter, the recursive call will eventually happen on an empty and a nonempty list – but we have no patterns which cover this possibility. Instead of dropping the elements of the nonempty lists (like the normal `zip` function), we encounter a pattern match exception.

6. What assumptions do the "making change" functions make about the variables `till` and `amt`? What could happen if these assumptions were violated?

> Most importantly, the `till` contains positive coin values in nonincreasing order. Having duplicate coin values wouldn't cause issues, but a larger coin value will never be used if it appears later in the list than a lower coin value. If any of the values is negative or zero, the function will not terminate – the amount will either increase or not change, but will never reach zero. If the `amt` is initially negative, the code terminates with the correct answer that no solution is possible, but it has to perform the whole algorithm before concluding that.

5. Sorting

5.1. Conceptual questions

1. You are given a long list of integers. Would you rather:

   a) Find a single element with linear search or sort the list and use binary search?

   > Finding the last element is linear in the worst case (the scenario being that it is the last element), sorting can generally be considered to be $O(n \log n)$ (e.g. using mergesort or quicksort), so for a single element sorting is wasteful.

   b) Find a lot of elements with linear search or sort the list and use binary search?

   > Finding k elements is $k \times O(n) = O(kn) \approx O(n^2)$ with linear search, but $k \times O(\log n) = O(k \log n) \approx O(n \log n)$ with binary search. The latter requires a preliminary step of sorting the list, which can be done in $O(n \log n)$ time, but this does not increase the

> worst-case complexity of the second method. Hence, if we are planning to look up a lot of elements in a list (e.g. if it models a database), it is heavily recommended to sort the list first and then use the much more efficient binary search.

c) Find all duplicates by pairwise comparison or sort the list and check for adjacent values?

> Pairwise comparison performs $O(n)$ comparisons for each element, giving quadratic complexity. Sorting is $O(n \log n)$, and checking for adjacent elements is linear, so sorting the list first is recommended.

2. Another sorting algorithm (*bubble sort*) consists of looking at adjacent pairs of elements, exchanging them if they are out of order and repeating this process until no more exchanges are possible. Analyse the time complexity of this approach.

> The best-case scenario is when the list is already sorted – the algorithm will not do anything to the list and terminate in linear time (contrast this with more efficient algorithms, which may nevertheless try sorting a sorted list – such as quicksort with a naive choice of the pivot). The worst case is a reverse-sorted list: every pass of the algorithm will bubble the head of the list up to the very end of the unsorted portion of the list. This takes $n+(n-1)+(n-2)+\cdots+1=\sum_{k=1}^{n} k$ steps, which equals $\frac{n(n-1)}{2}$ – multiplying this gives us a quadratic runtime in the worst case.

5.2. Exercises

3. Implement bubble sort in OCaml.

> The easiest approach is to split the algorithm into two functions: a single sweep across the list which performs all possible exchanges, and a recursive sorting function that performs the sweep until the list is sorted. The former uses nested pattern matching to compare the first two elements, and the latter checks if the sweeping changed anything – if not, the sort is complete.

```
let rec sweep = function
  | []            -> []
  | [x]           -> [x]
  | (x1::x2::xs) -> if x1 > x2 then x2::sweep (x1::xs)
                              else x1::sweep (x2::xs)

let rec bubblesort l = let l' = sweep l
                        in if l <> l' then bubblesort l' else l
```

5.3. Optional questions

4. Another sorting algorithm (selection sort) consists of looking at the elements to be sorted, identifying and removing a minimal element, which is placed at the head of the result. The tail is obtained by recursively sorting the remaining elements.

a) State, with justification, the time complexity of this approach.

> To find the minimum element in an unsorted list, we cannot do better than checking the entire list. We need to do this for every element in the unsorted part of the list, so there will be $n+(n-1)+(n-2)+\cdots+1$ comparisons, which is characteristic of an $O(n^2)$ algorithm.

b) Implement selection sort using OCaml.

> This solution uses two *mutually recursive* functions which call each other (and therefore indirectly call themselves recursively). As before, we separate the algorithm into a selection step and a sorting step that repeatedly performs the selection of the minimum element. The `select` function traverses the list argument, moving all the elements to the third argument (which acts as an accumulator) except the minimum one – this is determined by keeping track of the "running minimum" in the `min` argument and comparing it to the head of the list. When the list is depleted and the minimum element is found, we mutually recursively sort the rest of the list while adding the minimum to the front. The outer `selection_sort` function kicks off the process by calling `select` with the tail and the head as the running minimum.

```
let rec selection_sort = function
  | [] -> []
  | x::xs ->
   let rec select min ys rest = match ys with
     | []    ->  min::selection_sort rest
     | y::ys -> if y < min then select y ys (min::rest)
                           else select min ys (y::rest)
   in select x xs []
```

6. Datatypes and trees

6.1. Conceptual questions

1. Examine the following function declaration. What does `ftree (1,n)` accomplish?

```
let rec ftree k = function
  | 0 -> Lf
  | n -> Br(k, ftree (2*k) (n-1), ftree (2*k+1) (n-1))
```

> The function builds a *full binary tree* of depth n, i.e. a binary tree that has the maximum number of elements for a given depth (the equal case for the inequality above arises for full binary trees). The labels are calculated from some initial root label – when it is initialised to 1, we get a full binary tree labelled with integers from 1 to 2^n-1.

6.2. Exercises

2. Write an function taking a binary tree labelled with integers and returning their sum.

> A simple recursive solution works well here.
>
> ```
> let rec sum = function
>   | Lf               -> 0
>   | Br (x, lt, rt) -> x + sum lt + sum rt
> ```

3. Give the declaration of an OCaml datatype for arithmetic expressions that have the following possible shapes: floats, variables (represented by strings), or expressions of the form

$$-E \quad \text{or} \quad E+E \quad \text{or} \quad E \times E$$

Hint: recall how expressions differ from statements, and how their characteristic structure could be captured as a data type. It's a lot simpler than it may seem!

> As we saw before, datatypes consist of a list of constructors with arguments. The different constructors are various *alternatives* for the shape of the datatype: a list is *either* empty *or* a cons cell, a tree is *either* a leaf *or* a branch, a vehicle is *either* a lorry *or* a car *or* a motorbike, etc. Each alternative can be associated with one or more types given as the constructor arguments. Hence you can think of every constructor as representing a product of types (which, in OCaml, is the product type `'a * 'b` itself), and a datatype is a disjoint union/sum of these products as constructors. This is why OCaml-style datatypes are called *algebraic datatypes*: they are simply sums of products of types. The type
>
> ```
> type ty = A of int * bool | B of string | C
> ```
>
> can be thought of as the algebraic type
>
> $$\texttt{ty} = (\texttt{int} \times \texttt{bool}) + \texttt{string} + \texttt{unit}$$
>
> where OCaml's `|` is interpreted as the type-theoretic sum $+$, the product `*` is the type-theoretic product $\times$, and the nullary constructor simply represents the unit type. Unlike in OCaml, the constructors do not have explicit names – all we care about is a very simplified representation of the type. If you are familiar with set theory, you can think of a type as a set, and the operators $+$ and $\times$ as the disjoint union and Cartesian product of sets.
>
> But this notation is a bit confusing in this case. The sum and product mentioned in the question are different from the type-theoretic sum and product: the operations in the question refer to the *syntax* of the expression language. Where the algebraic view of datatypes comes in helpful is decoding the "specification" of the type. The question lists the alternatives for what shape a value of our `expr` type might take: it should be *either* a float, *or* a variable, *or* a negation of an expression, *or* sum of two expressions, *or* product

of two expressions. From this we know that we will need five alternatives, i.e. five data constructors. Now to deduce the number and type of arguments to the constructors, we consider what we need to construct an appropriate value. To get a real expression (in our syntax), we need an argument of the type `real`. Similarly, to get a variable expression, we need a string (as per the question). To get a negated expression, we need an inner subexpression – as expected, `expr` is a recursive type. Similarly, to create the sum and product of two expressions, we need the two subexpressions. Putting everything together, we have the algebraic type

$$\texttt{expr} = \texttt{float} + \texttt{string} + \texttt{expr} + (\texttt{expr} \times \texttt{expr}) + (\texttt{expr} \times \texttt{expr})$$

where each term of the sum corresponds to the constructor, and the terms themselves are the arguments to the individual constructors. In OCaml syntax we can also give the constructors arbitrary names, so the datatype would look like:

```
type expr = Float of float
          | Var   of string
          | Neg   of expr
          | Plus  of expr * expr
          | Times of expr * expr
```

4. Continuing the previous exercise, write a function `eval : expr -> float` that evaluates an expression. If the expression contains any variables, your function should raise an exception indicating the variable name.

In the previous question we defined the *syntax* of our expression language: the formal grammar that describes the structure of the expressions that we write. However, a value of type `expr` is nothing more than a nested hierarchy of the constructors which – on their own – don't "do" or mean anything. For example, we know that the term

```
let e = Plus (Neg (Times (Float 2.0, Float 3.0)), Float 8.0);
```

represents the mathematical expression $-(2 \times 3) + 8$, but `e` is not *equal* to the expression `Float 2.0`. In fact, if we didn't have descriptive names for the constructors (which we can choose arbitrarily), we might not even know what an expression represents. What we want is to give a meaning to our expression language, which in computer science is usually called the *semantics* for a particular grammar or syntax. There are multiple ways to give a semantics to some syntax (all of which will be covered in the Computer Science course), but in our case we want an *evaluation function*, which reduces the expression to an irreducible value (this is related to the approach of denotational semantics). Writing evaluation functions for recursive expression languages is really simple and elegant, as the implementation closely follows the inductive definition of the `expr` type.

We begin with declaring an exception for expressions with free variables (so-called *open terms*): this is supposed to return the name of the variable, so the exception will have a string argument. Then, we define the eval : expr -> float function by pattern-matching on the expression, and essentially translate the constructors to their corresponding mathematical operation – the Neg constructor becomes negation, the Plus becomes the operator +., etc. These operators work on float values, so we need to convert the subexpressions of type expr into float – but this is exactly what our eval function does, so we just call it recursively on the subexpressions. In the end, we get a function which, given any expr value, converts it into an OCaml arithmetic expression, which then gets evaluated to the corresponding floating-point number.

```
exception Variable of string

let rec eval = function
  | (Float f)         -> f
  | (Var v)           -> raise (Variable v)
  | (Neg e)           -> -. (eval e)
  | (Plus  (e1, e2)) -> eval e1 +. eval e2
  | (Times (e1, e2)) -> eval e1 *. eval e2;
```

Calling eval on the expression e above now correctly evaluates to the float 2.0.

As simple as this looks, it has very deep implications and the technique is used all over computer science – in fact, OCaml itself can be defined by giving a datatype specifying its syntax, and an evaluation function which determines the semantics.

6.3. Optional questions

5. Prove the inequality involving the depth and size of a binary tree t from Page 53.

$$\forall \text{ trees } \mathtt{t}.\ \text{count}(\mathtt{t}) \leq 2^{\text{depth}(\mathtt{t})} - 1$$

Hint: One way to do this is with a generalisation of mathematical induction called *structural induction*, where you analyse the *shape* (top-level constructor) of the tree t, and prove the property for the base case (leaf) and recursive case (branch). You can also try standard mathematical induction on some numerical property of the tree, but be careful with that you are assuming and what you are proving!

The proof is doable via mathematical induction on the height of the tree, but to make it fully rigorous, you need to be careful with your proof goals and assumptions. The tree may not be balanced so in the inductive case ($\text{depth}(\mathtt{t}) = k + 1$) you will only be able to apply the hypothesis to one of the subtrees (of depth k), but not the other. You can overcome this by doing *strong induction* (see the Discrete Maths course), or by proving a stronger theorem for perfectly balanced binary trees and deriving the above theorem as a corollary.

Instead, we will try a more general proof technique that can be applied to any inductively defined datatype (in fact, mathematical induction is just structural induction on the type `type nat = zero | succ of nat`). To use structural induction to prove a predicate about every value of some type, we prove it for the base cases of the type, then prove it for the general cases, assuming that the predicate holds for the data arguments. In this case, the base case of the tree `t` is that it is a `Lf`. Then,

$$\text{count}(\texttt{Lf}) = 0 \le 2^{\text{depth}(\texttt{Lf})} - 1 = 2^0 - 1 = 0$$

For the inductive case, assume that `t = Br (v, lt, rt)` and the predicate holds for the two subtrees, i.e.

$$\text{count}(\texttt{lt}) \le 2^{\text{depth}(\texttt{lt})} - 1 \qquad \text{and} \qquad \text{count}(\texttt{rt}) \le 2^{\text{depth}(\texttt{rt})} - 1$$

Then, we have

$$\begin{aligned}\text{count}(\texttt{Br (v, lt, rt)}) &= 1 + \text{count}(\texttt{lt}) + \text{count}(\texttt{rt}) \\ &\le 1 + (2^{\text{depth}(\texttt{lt})} - 1) + (2^{\text{depth}(\texttt{rt})} - 1) \\ &= 2^{\text{depth}(\texttt{lt})} + 2^{\text{depth}(\texttt{rt})} - 1\end{aligned}$$

where we used the two induction hypotheses to replace the count of the subtrees with the exponential of their depth. Now, assuming that the left subtree is deeper (without loss of generality), we have that

$$2^{\text{depth}(\texttt{Br (v, lt, rt)})} = 2^{1+\text{depth}(\texttt{lt})} = 2 \times 2^{\text{depth}(\texttt{lt})}$$

Going back to our previous inequality, we have that $2^{\text{depth}(\texttt{rt})} \le 2^{\text{depth}(\texttt{lt})}$, so

$$2^{\text{depth}(\texttt{lt})} + 2^{\text{depth}(\texttt{rt})} - 1 \le 2 \times 2^{\text{depth}(\texttt{lt})} - 1 < 2 \times 2^{\text{depth}(\texttt{lt})}$$

which establishes the required inequality

$$\text{count}(\texttt{Br (v, lt, rt}) \le 2^{\text{depth}(\texttt{Br (v, lt, rt)})}.$$

6. Give a declaration of the data type `day` for the days of the week. Comment on the practicality of such a datatype in a calendar application.

The `day` datatype is a good example of an enumeration type: we simply give all possible values of the type, without any constructor arguments. As we have seven days in the week, we will have seven constructors.

```
type day = Monday | Tuesday  | Wednesday | Thursday
         | Friday | Saturday | Sunday
```

The benefits of this representation is that it is clear and unambiguous: a function taking a `day` needs to handle these seven possibilities and only them. If we represented days with normal integers, we would need to worry about out-of-bounds numbers, representation

> conventions (e.g. is Monday 0 or 1, etc.) and the function types would be less descriptive (e.g. the function dayFromDate in the next exercise would just have the type int -> int -> int -> int). However, integers admit arithmetic operations, such as adding or subtracting days (though we need to check bounds), which would need to be explicitly defined for our day datatype.

7. Write a function dayFromDate : int -> int -> int -> day which calculates the day of the week of a date given as integers. For example, dayFromDate 2020 10 13 would evaluate to Tuesday (or whatever encoding you used in your definition of day above). *Hint*: Using Zeller's Rule might be the easiest approach.

> Zeller's Rule gives a simple arithmetic method of computing the day of the week for a given date. It converts the day, month, century and year to number, then computes the number of the day in the week that those numbers correspond to. Using our algebraic datatype representation, we need to convert this number into a value of type day – this can be done by pattern-matching or other ways (e.g. indexing into a list of those days).
>
> The algorithm first converts the dates into numbers, following specific rules that make the arithmetic simpler (e.g. months are shifted, numbering March as 1 and February as 12 – this way, the leap day is always at the end of the "year"). Finally, these numbers are plugged into the appropriate expression to get the number of the weekday, which is then converted to a day. We use the OCaml's integer division operator to implicitly take the integer part of the quotients, as described in the algorithm.

```
let numToDate = function
  | 0 -> Sunday    | 1 -> Monday
  | 2 -> Tuesday   | 3 -> Wednesday
  | 4 -> Thursday | 5 -> Friday
  | 6 -> Saturday

let dayFromDate yr mnth dy =
  let c = yr / 100
  and d = yr mod 100 - (if mnth < 3 then 1 else 0)
  and m = (mnth - 3) mod 12 + 1
  and k = dy
  in numToDate ((k + ((13 * m - 1) / 5) + d
                 + (d / 4) + (c / 4) - 2*c) mod 7)
```