p2 q9
p11 q13
LCP

2001

Software Engineering II (full question, Paper 2 and Diploma)

Describe, with examples, how the choice of programming language, programming tools and libraries can affect the reliability of the software developed using them. [5 marks]

Considering the following pair of ML function declarations:

```
fun takew p [] = []
  | takew p (x::xs) = if p x then x :: takew p xs else [];

fun dropw p [] = []
  | dropw p (x::xs) = if p x then dropw p xs else x::xs;
```

Prove `(takew p xs) @ (dropw p xs) = xs` using induction. (Assume that function p always terminates.) [8 marks]

You have been asked to specify some banking software. A bank account has a *balance* and an overdraft *limit*, subject to the constraints $limit \geqslant 0$ and $balance + limit \geqslant 0$.

Write a Z schema to specify the state of a bank account. [2 marks]

Write a Z schema for the operation to withdraw a given positive *amount* from the account. [5 marks]

Model Answer

The first part is bookwork from Lecture 3. A language like Ada or Modula-3 can promote reliability by enforcing strong type-checking and minimizing the use of explicit memory allocation. The C programmer can compensate by using diagnostic tools and debugging harnesses (e.g. Electric Fence) to check for compile-time and run-time errors. Bad features of libraries include functions with confusing and complicated calling sequences, and functions that omit safety checks e.g. that the supplied buffer does not overflow. Good libraries not only avoid such problems but can actually replace large parts of the developed product; parser generators like Yacc are typical.

The base case of the induction is `(takew p []) @ (dropw p []) = []`. Both sides collapse to `[]`.

In the induction step we assume `(takew p xs) @ (dropw p xs) = xs` and prove `(takew p (x::xs)) @ (dropw p (x::xs)) = (x::xs)`. There are two cases, depending on whether or not `p x` is true. If it is then we have

```
(takew p (x::xs)) @ (dropw p (x::xs))
= (x :: takew p xs) @ (dropw p xs)
```

```
= x :: (takew p xs) @ (dropw p xs)
= x::xs,
```

using the induction hypothesis in the last step. If `p x` is false then we have

```
(takew p (x::xs)) @ (dropw p (x::xs)) = [] @ (x::xs) = x::xs.
```

The state of a bank account is given by this schema, with the constraints given as invariants. We assume that *MONEY* is a numeric type.

Account

$$balance, limit : MONEY$$

$$limit \geqslant 0$$
$$balance + limit \geqslant 0$$

The predicate of the *Withdraw* schema tests for a positive amount. The *balance* is reduced while the *limit* remains constant. We do not need to test for sufficient funds because the schema inherits the *Account* invariants, which express this constraint already: from $balance' + limit' \geqslant 0$ and the equations below we get $balance - amount + limit \geqslant 0$.

Withdraw

$$\Delta Account$$
$$amount? : MONEY$$

$$amount > 0$$

$$balance' = balance - amount$$
$$limit' = limit$$