About the correctness of a few SASL programs

Shortly after my arrival in Austin, Texas, I mentioned the longest-upsequence problem. A few -warm!- days later, dr. Hamilton Richards offered me four solutions for my consideration. I shall render his solutions in the official SASL syntax - in spite of my objections to it — and in order to save dr. Richards from possible blame I would like to stress that the inefficiency of the first solution was quite deliberate.

* * *

Given a sequence M of integers, find a longest ascending subsequence (i.e. upsequence) of M.

First solution: Generates all subsequences, filters out the non-ascending ones, and chooses the longest.

```
longest (filter ascending (subs M))
where
  subs ()    = ((),)
  subs (p:x) = {p:subx ; subx ← subsx} ++ subsx
               where subsx = subs x

  ascending ()       = true
  ascending (a,)     = true
  ascending (a:b:x)  = a ≤ b ∧ ascending (b:x)

  filter c ()    = ()
  filter c (a:x) = c a → a: filter c x ;
                   filter c x
```

(NB. The semicolon plays the rôle of else.)

```
  longest (a,)  = a
  longest (a:x) = longer a lx → a ; lx
                  where lx = longest x
                        longer x ()       = true
                        longer () y       = false
                        longer (a:x) (b:y) = longer x y
```

* * *

Let me add some legenda to the above. With x a sequence of integers, the value of subs x is a list of all subsequences of x. Between the braces subx is the bound

variable ; it is of type subsequence and the "←" indicates that it ranges over the elements of subsx . (The list formation denoted by the shorthand of the braces is order preserving. If subx1 occurs before subx2 in subsx , p:subx1 occurs before p:subx2 in the whole brace expression.) The ++ denotes concatenation. It is clear that subs (p:x) contains twice as many elements as subs x : all subsequences of M, including the empty one, are generated.

The argument of ascending is of type integer sequence. (I find myself reserving the term sequence for what is syntactically a list of integers. Thus I avoid having to talk about lists of lists.) The function ascending – which is of type boolean – is separately defined for the empty sequence, for the 1-element sequence, and for the sequence of 2 or more elements. The last one is recursive and recursion ends with a 1-element sequence as argument – viz. when x = ( ) – . Our program, however, requires ascending ( ) to be defined as well (one way or the other) .

The function filter is a very common one. (I suspect it is in the library, for Richards had omitted its definition.) The first argument is a boolean function that accepts as argument the elements of filter's second argument, which is a list.

The value of longest applied to a non-empty list of sequences is a sequence; longer is a boolean function defined on two sequences , and defined provided at least one of its arguments is of finite length . We note that, M being finite, all our sequences are finite. Denoting the length of a sequence by # , longer x y = (# x ≥ # y).

Note that the names subsx and lx have only been introduced for the sake of efficiency; automatic introduction of them requires the mechanized detection of common subexpressions.

This program is very unsophisticated, so much so, as a matter of fact, that I feel that a more elaborate correctness argument would to my taste be wasted. I felt obliged, however, to state explicitly the type of all expressions, and of all arguments, and to check that all functions are applied

to arguments of the appropriate type and that the functions are defined for all arguments to which they may be applied.

A quite likely error would be to forget to define "ascending ()", the more so since its value is irrelevant for $M \neq ()$.

Another likely error is to define erroneously subs () = (). The list of subsequences of an empty sequence is not the empty list, but a 1-element list containing the empty sequence as its only element. So much for the first solution.

* * *

Second solution: Generates only upsequences.

```
longest (ups M)
where
  ups ()    = ((),)
  ups (p:x) = {p:upx ; upx ← upsx ; upx=()| hd upx ≥ p} ++ upsx
              where upsx = ups x
```

* * *

Let me add some legenda first. Here list formation is filtered by the condition upx=()| hd upx ≥ p (where the vertical stroke stands for cor).

There are now two ways open to us. One of them is to read the above definition carefully and to say to ourselves yes, ups M is the list of all upsequences of M. That would satisfy me, but for the sake of argument we shall pursue the second path and remember our first solution. Because longest is a function and our first solution is correct, so is our second one provided we can prove

(0) ups M = filter ascending (subs M).

Let us try. We first investigate the case M = ()

```
    filter ascending (subs ())
=   { def. of subs}
    filter ascending ((),)
=   {def. of filter}
    ascending () → (): filter ascending () ;
    filter ascending ()
```

```
= { ascending ( ) = true}
  ( ) : filter ascending ( )
= { definition of filter}
  ( ) : ( )
= { a : ( ) = (a,)}
  (( ),)
```

Note. I believe that SASL does not require the outer parentheses.

Aside. I am not so certain I like SASL's special notation "x," for the one-element list containing only x. (End of Aside.)

Now the case M = p:x ; for the sake of brevity we shall eliminate the identifiers subsx and upsx .

```
  filter ascending (subs (p:x))
= {def. of subs}
  filter ascending ({p:subx ; subx ← subs x} ++ subs x)
= { lemma 0 about filter}
  filter ascending {p:subx ; subx ← subs x}
    ++ filter ascending (subs x)
= {lemma 1 about filter and list formation}
  {p:subx; subx ← subs x ; ascending (p:subx)}
    ++ filter ascending (subs x)
= {definition of ascending}
  {p:subx: subx ← subs x ; subx = ( ) | (p ≤ hd subx ∧ ascending subx)}
    ++ filter ascending (subs x)
= {because ascending ( ) = true}
  {p:subx: subx ← subs x ; (subx = ( ) | p ≤ hd subx ) ∧ ascending subx}
    ++ filter ascending ( subs x)
= { lemma 2 about filter and list formation}
  { p:subx : subx ← filter ascending (subs x); subx = ( ) | p ≤ hd subx}
    ++ filter ascending (subs x)
```

Lemma 0: filter c (A ++ B) = filter c A ++ filter c B

Lemma 1: filter c {f x; x ← y} = {f x; x ← y ; c (f x)}

Lemma 2 : {f x ; x ← y ; b (f x) ∧ c x} =
{f x; x ← filter c y ; b (f x)}

Realizing that subx is a dummy we see that filter ascending (subs (p:x)) satisfies the defining equation for ups (p:x) , so (0) has been proved. To say that I liked the above exercise would be an exaggeration.

* * *

Third solution: For each element of M, generates only the longest upsequence beginning with that component.

```
longest (ups' M)
where
  ups' ( )   = ((),)
  ups' (p:x) = (p: longest {up; up ← upx; up=( )| hd up ≥ p}): upx
               where upx = ups' x
```

* * *

This time further legenda are not necessary. It is undoubtedly possible to derive the third solution from the second one in very much the same way as we have derived the second solution from the first one. This time we would have to exploit the properties of longest such as

$$\text{longest}\ (A \mathbin{+\!\!+} B) = \text{longest}\ ((\text{longest}\ A) : B)$$

and

$$\text{longest}\ \{p : up; \ldots\} = p : \text{longest}\ \{up; \ldots\}$$

I shall not do it, nor prove above properties of longest, since I don't expect the exercise to be instructive.

We note that inventing new identifiers has become a burden for the author: he reuses upx in a meaning very different from the one it had in the second solution.

The number of subsequences to be investigated has been reduced drastically: from 2^N in the first solution, via N^2 in the second solution to N in the third one.

I find it hard to estimate the third solution's time complexity. My guess is that longest will in general be invoked order N^2 times, and that the algorithm is therefore of order N^3. The nature of longest is such that I see very little gain from lazy evaluation. (For lack of experience and knowledge above estimations are hardly "educated guesses".)

* * *

Fourth solution: Restricts exploration to those upsequences that are not included in longer upsequences.

```
longsift (ups" M)
where
  ups" ()    = ((),)
  ups" (p:x) = (p:longsift {up; up ← upx; up=() | hd up ≥ p}): upx

  longsift = longest • sift
  sift = sift' PosInf

  sift' min ()     = ((),)
  sift' min (a:x) =  a=() → sift' min x ;
                     p ≥ min → sift' min x ;
                     a : sift' p x
                     where (p:r) = a
```

* * *

Legenda: the • in the definition of longsift denotes functional composition, which needs to be indicated explicitly because in SASL functional application is defined to be left-associative. The name longsift could have been avoided at the price of another parenthesis pair

```
(p: longest (sift{up; up←upx; up=() | hd up ≥ p}))
```

and similarly in the first line.

The whole purpose of the introduction of sift is to reduce the number of (expensive) invocations of longest. One has to prove that

```
longest {up; up←upx;...} = longest (sift {up; up←upx;...})
```

but now we are reaching the stage of the coding trick since now we have to take into account the order in which the sequences occur in upx plus the fact that the list formation with the braces is —contrary to what the braces from set notation suggest!— order-preserving. Process sift drops those sequences of which it can cheaply be established that they are not a longest upsequence from the set considered, viz. by comparing their first elements with the minimum element of M "to their left and encountered so far". (I refer the interested reader to EWD824, of which Richards was unaware.)

* * *

For the sake of completeness I mention that Mark Scheevel and Bill Kelley have written SASL programs for a longest upsequence and the maximum upsequence length, probably the first and certainly the latter being of order N^2 – and not of order $N \cdot \log N$ since the binary search of the imperative program had to be replaced by a linear one —. They used foldl

```
foldl op r ()     = r
foldl op r (a:x) = foldl op (op r a) x
```

which is nothing but the SASL coding of a for-statement — with op as repeatable statement and r as the state-variable — iterating from left to right over the elements of a finite list.

```
maxup M
where
maxup = length • maxupsequence

maxupsequence = foldl add ()

add () n = n:()
add (m:mm) n = n ≥ m → n:m:mm ;
               trickle m mm
               where
               trickle mk (mk1:mm) = n ≥ mk1 → n:mk1:mm ;
                                     mk : trickle mk1 mm
               trickle mk () = n:()
```

maxupsequence generates a list of the k "best" upsequence heads; the best ith upsequence is the upsequence of length i with the smallest head.

I was pleased to see their use of :() instead of the comma to construct a singleton list. Note, however, their reuse of the identifier mm .

* * *

From all the above, pg. EWD825-3 in particular left me with very mixed feelings. On the one hand I had recently learned to appreciate such proofs that consist of a chain of simple manipulations, on the other hand I knew I was disgusted. Why?

There is something profoundly wrong with the cost/performance ratio. You see, while I have recently used a similar style for very concise proofs of surprising theorems, this time the theorem was obvious to start with, a circumstance that makes the whole exercise insipid.

Besides that, Lemma 2 made me suspicious: it is just a funny way of rewriting the well-known formula

$$[Q \wedge (\underline{E}i :: P.i) = (\underline{E}i :: Q \wedge P.i)]$$

from predicate calculus.

Furthermore there is a whole class of functions f

f: bag of elements → element

with the property $f(B0 \circledast B1) = f(fB0 \circledast fB1)$, and it hurts to have to write this down in the very special form

longest (A ++ B) = longest (longest A ++ longest B) .

It has most definitely nothing to do with the concatenation which is only used to add bags when bags are represented by lists.

The set A of all subsequences of M is defined by

$$(\underline{A}x :: x \ \underline{in} A = x \text{ is subsequence of } M) .$$

With subs as defined on pg. EWD825-0, we can prove that subs M represents a particular listing of the elements of A; it is probably a mistake to take subs M as A's definition and to derive all relevant properties of A from the way in which subs has been defined. My manipulations on pg. EWD825-3 are very close to program development by transformations, a technique that seems to invite making that mistake.

Burroughs Corporation
12201 Technology Blvd
AUSTIN, Texas 78759
U.S.A.

31 May 1982
prof.dr. Edsger W. Dijkstra
Burroughs Research Fellow