COM2001 Assignment 2: Theory Daniel Marshall (160152953)

(1.) We will prove that the given equality holds for all finite defined values str of type Stream T.

First we need a lemma, so let P(str) be the statement that sLen str == sLen (sPop (Append str x)) for any x; we will prove that P(str) holds for all finite defined Streams str using structural induction.

For the base case, we need to show that P(None) holds. The left hand side gives:

sLen None
= Zero by [sLen.0].

The right hand side gives:

sLen (sPop (Append None x))
= sLen None by [sPop.1]
= Zero by [sLen.0].

Thus, the base case holds. Now, for the inductive step, we need to show that assuming P(k) holds, we must have P(Append k y) also. Considering P(Append k y), the left hand side gives:

sLen (Append k y)
= Succ (sLen k) by [sLen.n].

The right hand side gives:

sLen (sPop (Append (Append k y) x))
= sLen (Append (sPop (Append k y)) x)) by [sPop.n]
= Succ (sLen (sPop (Append k y))) by [sLen.n]

= Succ (sLen k), by [assumed].

As both sides evaluate to Succ (sLen k), the inductive step holds, and we have proven the lemma holds for all finite defined values of str by structural induction.

Now, let Q(n) be the statement that sLen str == sLen (sRev str) whenever sLen str = n; this is equivalent to the given statement, and we can prove it by structural induction over Nat.

For the base case, we need to show that Q(Zero) holds. In this case, sLen str = Zero, so by [sLen.0] we must have str == None. Then the right hand side gives:

sLen (sRev None)
= sLen None   by [sRev.0]
= Zero   by [sLen.0].

Thus, the base case holds. Now, for the inductive step, we need to show that assuming Q(k) holds, we must have Q(Succ k) also. Considering Q(Succ k), we see that sLen str = Succ k, so by [sLen.n] we have that str == Append s x for some x, where sLen s = k. Then the left hand side gives:

sLen str
= sLen (Append s x)   by [definition]
= Succ (sLen s)   by [sLen.n]
= Succ k   by [definition].

The right hand side gives:

sLen (sRev str)

$= \text{sLen}\ (\text{sRev}\ (\text{Append}\ s\ x))$ by [definition]

$= \text{sLen}\ (\text{Append}\ (\text{sRev}\ (\text{sPop}\ (\text{Append}\ s\ x))\ \ \text{sTop}\ (\text{Append}\ s\ x)))$ by [sRev.n]

$= \text{Succ}\ (\text{sLen}\ (\text{sRev}\ (\text{sPop}\ (\text{Append}\ s\ x))))$ by [sLen.n]

Now, by the lemma proved earlier, $K = \text{sLen}\ s = \text{sLen}\ (\text{sPop}\ (\text{Append}\ s\ x))$, so we have:

RHS $= \text{Succ}\ (\text{sLen}\ (\text{sPop}\ (\text{Append}\ s\ x))$ by [assumed]

$= \text{Succ}\ (\text{sLen}\ s)$ by [lemma]

$= \text{Succ}\ k$ by [definition].

As both sides evaluate to Succ k, the inductive step holds, and so by structural induction the given statement is true for all finite defined values str of type Stream T.

(2) From the Haskell compiler, and accounting for the monomorphism restriction, we know that the correct type is:

$$h :: \text{Num } a \Rightarrow ((\text{Either Bool } a \to b) \to (c \to \text{Either } c \text{ Int}) \to t) \to t.$$

As $(\backslash x \to x\ f\ g)$ is an anonymous function, we can start by assuming that x has type $((\text{Either Bool } a \to b) \to (c \to \text{Either } c \text{ Int})) \to t$.

Now we use the function application rule:

From

$x :: ((\text{Either Bool } a \to b) \to (c \to \text{Either } c \text{ Int})) \to t$

$f :: \text{Num } a \Rightarrow \text{Either Bool } a \to b$

Deduce

$x\ f :: \text{Num } a \Rightarrow (c \to \text{Either } c \text{ Int}) \to t$

Next, we use the function application rule again:

From

$x\ f :: \text{Num } a \Rightarrow (c \to \text{Either } c \text{ Int}) \to t$

$g :: c \to \text{Either } c \text{ Int}$

Deduce

$x\ f\ g :: \text{Num } a \Rightarrow t.$

Finally, we can use the abstraction rule:

From

$(x :: (\text{Either Bool } a \to b) \to (c \to \text{Either } c \text{ Int}) \to t)$ implies $(x\ f\ g :: \text{Num } a \Rightarrow t)$

Deduce

$(\backslash x \to x + g) :: $ Num a => ((Either Bool a -> b) -> (c -> Either c Int) -> t) -> t

And so we have:

h :: Num a => ((Either Bool a -> b) -> (c -> Either c Int) -> t) -> t.

③ For correctness, we need to prove that whenever diff xs ys halts, the result is equal to $\delta(xs, ys)$, and for total correctness, we need to prove that diff xs ys halts for any finite defined lists xs and ys.

First, consider the base case xs == ys == [ ]. Then diff xs ys = 0, and δ(xs, ys) = length ([ ]) − length ([ ]) = 0 also, so this case holds.

Next, consider the case ys == [ ]. Then diff xs ys = length xs, and δ(xs, ys) = length xs − length ([ ]) = length xs also, so this case holds.

We also have the case xs == [ ]. Then diff xs ys = − (length ys), and δ(xs, ys) = −δ(ys, xs) = − (length ys − length ([ ])) = − (length ys), so this case holds as well.

Now, the main case we need to prove is that diff list1 list2 = δ(list1, list2) when neither list1 or list2 is empty; that is, list1 == x:xs and list2 == y:ys.

In this case, if length (x:xs) < length (y:ys), then δ(x:xs, y:ys) = − δ(y:ys, x:xs) = − (length (y:ys) − length (x:xs)) = length (x:xs) − length (y:ys), and if length (x:xs) ≥ length (y:ys), then δ(x:xs, y:ys) = length (x:xs) − length (y:ys) also.

As well as this, we have that length (x:xs) − length (y:ys) = (length xs + 1) − (length ys + 1) = length xs − length ys, so in all cases δ(x:xs, y:ys) = length xs − length ys.

If length (x:xs) < length (y:ys) and xs is empty, we have diff x:xs y:ys = diff xs ys = − (length ys) = length ([ ]) − length ys = δ(x:xs, y:ys). If xs is not empty, then neither xs or ys is empty and diff x:xs y:ys = diff xs ys, so we repeat.

If length $(x:xs) \geq$ length $(y:ys)$ and both xs and ys are empty, then diff $x:xs$ $y:ys$ = diff xs ys = 0 = length ([ ]) - length ([ ]) = $\delta(x:xs, y:ys)$. If only ys is empty, then diff $x:xs$ $y:ys$ = diff xs ys = length xs = length xs - length ([ ]) = $\delta(x:xs, y:ys)$ also. If neither xs or ys is empty, then diff $x:xs$ $y:ys$ = diff xs ys and we repeat as before.

Now, as $x:xs$ and $y:ys$ are finite defined lists, this process will always end in either xs or ys or both becoming empty, so as diff $x:xs$ $y:ys$ $= \delta(x:xs, y:ys)$ in every possible case, the program is proved correct.

Also, since it always halts as stated, the program is totally correct as well.

(4) First we need to create functions sPopT :: Stream a → Int and sTopT :: Distinguished a => Stream a → Int, as we will need them later when creating sRevT. So we have:

```
sPopT (None) = 1 + T(None)          -- cost: case
             = 1 + 0                -- cost: const
             = 1                    -- arithmetic

sPopT (Append None _) = 1 + T(None)     -- cost: case
                      = 1 + 0           -- cost: const
                      = 1               -- arithmetic

sPopT (Append s' x) = 1 + T(Append (sPop s') x)     -- cost: case
                    = 1 + sPopT(s')                 -- cost: const
                    = 1 + sLen(s')                  -- arithmetic

sTopT (None) = 1 + T(special)     -- cost: case
             = 1 + 0              -- cost: const
             = 1                  -- arithmetic

sTopT (Append None x) = 1 + T(x)     -- cost: case
                      = 1 + 0        -- cost: const
                      = 1            -- arithmetic

sTopT (Append s' _) = 1 + T(sTop s')     -- cost: case
                    = 1 + sTopT(s')      -- cost: const
                    = 1 + sLen(s')       -- arithmetic
```

Now, using these functions, we can create the function sRevT :: Distinguished a => Stream a → Int. We have:

```
sRevT (None) = 1 + T(None)     -- cost: case
```

$= 1 + 0$ -- cost: const

$= 1$ -- arithmetic

$\text{sRevT (Append None \_)} = 1 + T(\text{Append None \_})$ -- cost: case

$= 1 + 0$ -- cost: const

$= 1$ -- arithmetic

$\text{sRevT}(s) = 1 + T(\text{Append (sRev } s') \; x')$ -- cost: case

$= 1 + T(\text{Append (sRev (sPop } s)) \text{ sTop } s)$ -- cost: const

$= 1 + \text{sRevT (sPop } s) + \text{sPopT}(s) + \text{sTopT}(s)$ -- cost: const

$= 1 + \text{sRevT (sPop } s) + 2(1 + \text{sLen } s)$ -- sPopT, sTopT

$= 3 + 2(\text{sLen } s) + \text{sRevT (sPop } s).$ -- arithmetic.

Next we can convert sRevT into a cost function sRevC, using sLen s as our cost parameter. Then:

$\text{sRevC}(0) = 1$

$\text{sRevC}(1) = 1$

$\text{sRevC}(n+1) = 3 + 2n + \text{sRevC}(n).$

As this formula is recursive, we can use the recurrence equation $f(n) = \frac{b}{2}n^2 + (c + \frac{b}{2})n + d$ to give:

$\text{sRevC}(n) = n^2 + 4n + 1.$

Therefore, sRev is tractable, with complexity $O(n^2)$.