COM2001 Advanced Programming Topics

Assignment 2 - Theory

Question 1.

Lemma 1. Prove that `sLen (sPop (Append s x)) == sLen s`

Let $P(s) \equiv$ `sLen (sPop (Append s x)) == sLen s`

***Lemma 1** Base Case.* $P($`None`$) \equiv$ `sLen (sPop (Append None x)) == sLen None`

```
LHS: sLen (sPop (Append None x))
     = sLen (None)                 -- [sPop.1]

RHS: sLen None
     == LHS
```

□

***Lemma 1** Step Case.* Assume $P(s) \equiv$ `sLen (sPop (Append s x)) == sLen s` is true. Prove $P($`Append s y`$)$, where `x` and `y` can be any values of same type.

```
LHS: sLen (sPop (Append (Append s y) x))
     = sLen (Append (sPop (Append s y)) x) -- [sPop.n]
     = Succ (sLen (sPop (Append s y)))     -- [sLen.n]
```

according to assumption, `sLen (sPop (Append s x)) == sLen s`

```
     = Succ (sLen s)                       -- assumption

RHS: sLen (Append s y)
     = Succ (sLen s)
     == LHS
```

□

To prove that `sLen str == sLen (sRev str)`, first

Let $P(str) \equiv$ `sLen str == sLen (sRev str)`

and then, it is required to prove that $P($`None`$)$, $P($`s`$)$ and $P($`Append s x`$)$ holds whenever `s` is a finite length of type `Stream a`.

Base Case. When `str` is `None`

```
LHS: sLen str
     = sLen None
     = Zero              -- by [sLen.0]

RHS: sLen (sRev str)
     = sLen (sRev None)
     = sLen (None)       -- by [sRev.0]
     = Zero              -- by [sLen.0]
     == LHS
```

Hence, the statement $P($`None`$)$ holds ☐

Step Case. Assume that the statement $P(s) \equiv$ `sLen s == sLen (sRev s)` holds. Prove that the statement $P($`Append s y`$)$, where `y` can be any values of type `a`, also holds.

```
LHS: sLen (Append s y)
     = Succ (sLen s)                             -- by [sLen.n]
```

according to **Lemma 1**, `sLen (sPop (Append s x)) == sLen s`, hence

```
     = Succ (sLen (sPop (Append s x)))           -- Lemma 1

RHS: sLen (sRev (Append s y))
     = sLen (Append (sRev s') y')                -- [sRev.n]

  where  s' = sPop (Append s y)
         y' = sTop (Append s y)

     = Succ (sLen (sRev (sPop (Append s y)))     -- [sLen.n]
```

Since `x` and `y` can be any values of the same type, and the output of the function `sPop` is type `Stream a`. According to our assumption, `sLen s == sLen (sRev s)`. Therefore, the statement `sLen (sPop (Append s x)) == sLen (sRev (sPop (Append s y))` should be true even if `x` does not equal to `y` because the length of the stream is not affected by the value of `x` or `y`. Hence, `LHS == RHS`

☐

According to the proofs above, it can be said that the statement $P(str) \equiv$ `sLen str == sLen (sRev str)` holds for all the finite defined values of `str`, providing that the result produced can be represented by a value in the set of finite defined values of `Nat`.

Question 2.

Proof. Given three functions, as defined below:

1. `f :: Num a => Either Bool a -> b`
2. `g :: c -> Either c Int`
3. `h = \x -> x f g`

The function `h` can be rewritten as

```
h x = x f g
```

Lemma 2. `h` is a function which takes a function `x` as argument, and return the result of the function `x` taking two functions `f` and `g` as argument.

Step 1. Assume `x :: a`
By Type Instantiation,

```
x :: a
x :: a {d -> e -> t / a}   -- type instantiation
x :: d -> e -> t
```

Step 2. Assume `f :: d`, where `d :: Num a => Either Bool a -> b`
By Function Application,

```
  x  :: d -> e -> t
  f  :: d                   -- assumed
-----------------------
 x f :: e -> t              -- function application
```

Step 3. Assume `g :: e`, where `e :: c -> Either c Int`
By Function Application,

```
  x f  :: e -> t
   g   :: e                 -- assumed
---------------------
 x f g :: t                 -- function application
```

Since `h x = x f g`, hence the type of the function `h` should be

`h :: (d -> e -> t) -> t`.

But it is assumed that `d :: Num a => Either Bool a -> b`, substituting `d` back to `h`,

`h :: Num a => ((Either Bool a -> b) -> e -> t) -> t`

It is also assumed that `e :: c -> Either c Int`, substituting `e` back to `h`,

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

□

To show that the proof above for the type of `h` is true, compare the result obtained above with the statement in **Lemma 2**.

In **Lemma 2**, it is stated that `h` takes a function `x` as argument, and return a result of type same as the result produced by the function `x` taking two arguments `f` and `g`.

Deduction:

1. The type of the result of function `h` and function `x` should be the same, which in this case is `t`. This can be seen from the definition of function `h`, as the output of `h` is equal to `x f g`.
2. According to **1**, then the type of `x f g` should be `t`.
3. According to **2**, and since `x` takes `f` and `g` as arguments, and the type of `x f g` is `t`, then the type of `x` should be
   `Num a => (Either Bool a -> b) -> (c -> Either c Int) -> t`
4. According to the deductions above, it is known that the return type of `h` is `t`, and `h` only take function `x` as argument, therefore the type of the function `h` should be
   `Num a => ((Either Bool a -> b) -> (c -> Either c Int) -> t) -> t`

The deductions above give the same result as the proof using the type inference rules. Therefore, it has been proved that the type of the function `h` is

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

Question 3.

Proof. To prove that `diff` is a 'totally correct' implementation of $\boldsymbol{\delta}$, it is required to prove that:

1. *Correctness*: For all finite defined lists, by giving `diff` and $\boldsymbol{\delta}$ the same input, `diff` should produced the same output as $\boldsymbol{\delta}$, and the output produced is the right answer.
2. *Total correctness*: The function `diff` will definitely halt.

1. *Proof of Correctness.* To prove the correctness of `diff`, let

$P($`xs`$) \equiv$ `diff(xs, ys) ==` $\boldsymbol{\delta}$`(xs, ys)`, for all finite defined `ys` of type `[b]`

Base Case 1. When $P($`[]`$) \equiv$ `diff([], ys) ==` $\boldsymbol{\delta}$`([], ys)`,

```
LHS: diff([], ys)
      = - (length ys)

RHS: δ([], ys)
      = - δ(ys, [])
      = - (length(ys) - length([]))
      = - length(ys)
      == LHS
```

□

Step Case 1. Assume that for all finite defined `ys` of type `[b]`, the statements

- $P($`s`$) \equiv$ `diff(s, ys) ==` $\boldsymbol{\delta}$`(s, ys)` is true if `length s >= length ys`
- $P($`s`$) \equiv$ `diff(s, ys) == -` $\boldsymbol{\delta}$`(ys, s)` is true if `length s < length ys`

When $P($`x:s`$) \equiv$ `diff(x:s, ys) ==` $\boldsymbol{\delta}$`(x:s, ys)` and `length x:s >= length ys`,

```
LHS: diff(x:s, ys)
  = diff(x ++ s, [] ++ ys)
  = diff(x, []) + diff(s, ys)
  = length x + diff(s, ys)                -- case definition
  = length x + δ(s, ys)                   -- assumption

RHS: δ(x:s, ys)
    = length(x:s) - length(ys)            -- δ definition
    = length(x) + length(s) - length(ys)
    = length(x) + δ(s, ys)                -- δ definition
    == LHS
```

When $P(x:s) \equiv$ `diff(x:s, ys) == - δ(ys, x:s)` and `length x:s < length ys`,

```
LHS: diff(x:s, ys)
  = diff(x ++ s, [] ++ ys)
  = diff(x, []) + diff(s, ys)
  = length x + diff(s, ys)                    -- case definition
  = length x - δ(ys, s)                       -- assumption

RHS: δ(x:s, ys)
    = - δ(ys, x:s)
    = - length(ys) + length(x:s)              -- δ definition
    = - length(ys) + length(x) + length(s)
    = length(x) - length(ys) + length(s)
    = length(x) - δ(ys, s)                    -- δ definition
    == LHS
```

□

It has been proved that the statements $P($`[]`$)$, $P($`s`$)$, and $P($`x:s`$)$ are true for all finite defined `ys` of type `[b]`. Therefore, it can be stated that the statement

$P($`xs`$) \equiv$ `diff(xs, ys) == δ(xs, ys)`, for all finite defined `ys` of type `[b]`

is true as well.

Now, let

$P($`ys`$) \equiv$ `diff(xs, ys) == δ(xs, ys)`, for all finite defined `xs` of type `[a]`

Base Case 2. When $P($`[]`$) \equiv$ `diff(xs, []) == δ(xs, [])`,

```
LHS: diff(xs, [])                 RHS: δ(xs, [])
    = length xs                       = length(xs) - length([])
                                      = length(xs)
                                      == LHS
```

□

Step Case 2. Assume that for all finite defined `xs` of type `[a]`, the statements

- $P($`s`$) \equiv$ `diff(xs, s) == δ(xs, s)` is true if `length xs >= length s`
- $P($`s`$) \equiv$ `diff(xs, s) == - δ(s, xs)` is true if `length xs < length s`

When $P($`y:s`$) \equiv$ `diff(xs, y:s) == δ(xs, y:s)` and `length xs >= length y:s`,

```
LHS: diff(xs, y:s)
  = diff([] ++ xs, y ++ s)
  = diff([], y) + diff(xs, s)
  = - (length y) + diff(xs, s)          -- case definition
  = - (length y) + δ(xs, s)             -- assumption

RHS: δ(xs, y:s)
    = length(xs) - length(y:s)          -- δ definition
    = length(xs) - (length(y) + length(s))
    = length(xs) - length(y) - length(s)
    = - length(y) + length(xs) - length(s)
    = - length(y) + δ(xs, s)            -- δ definition
    == LHS
```

When $P($`y:s`$) \equiv$ `diff(xs, y:s) == -` $\boldsymbol{\delta}$`(y:s, xs)` and `length xs < length s`,

```
LHS: diff(xs, y:s)
  = diff([] ++ xs, y ++ s)
  = diff([], y) + diff(xs, s)
  = - (length y) + diff(xs, s)           -- case definition
  = - (length y) - δ(s, xs)              -- assumption

RHS: δ(xs, y:s)
    = - δ(y:s, xs)                       -- δ definition
    = - length(y:s) + length(xs)         -- δ definition
    = - length(y) - length (s) + length(xs)
    = - length(y) - δ(s, xs)             -- δ definition
    == LHS
```

$\square$

It has been proved that the statements $P($`[]`$)$, $P($`s`$)$, and $P($`y:s`$)$ are true for all finite defined `xs` of type `[a]`. Therefore, it can be stated that the statement

$P($`ys`$) \equiv$ `diff(xs, ys) ==` $\boldsymbol{\delta}$`(xs, ys)`, for all finite defined `xs` of type `[a]`

is true as well.

Since it has been proved that

1. For all finite defined `ys` of type `[b]`, $P($`xs`$)$ is true
2. For all finite defined `xs` of type `[a]`, $P($`ys`$)$ is true

Hence, the *correctness* of `diff` has been proved, and `diff` is a correct implementation of the function $\boldsymbol{\delta}$. $\square$

2. Proof of Total Correctness. The *correctness* of `diff` has been proved. To prove that `diff` is a totally correct implementation of $\boldsymbol{\delta}$, it is required to show that `diff` is correct, and `diff` will definitely halt.

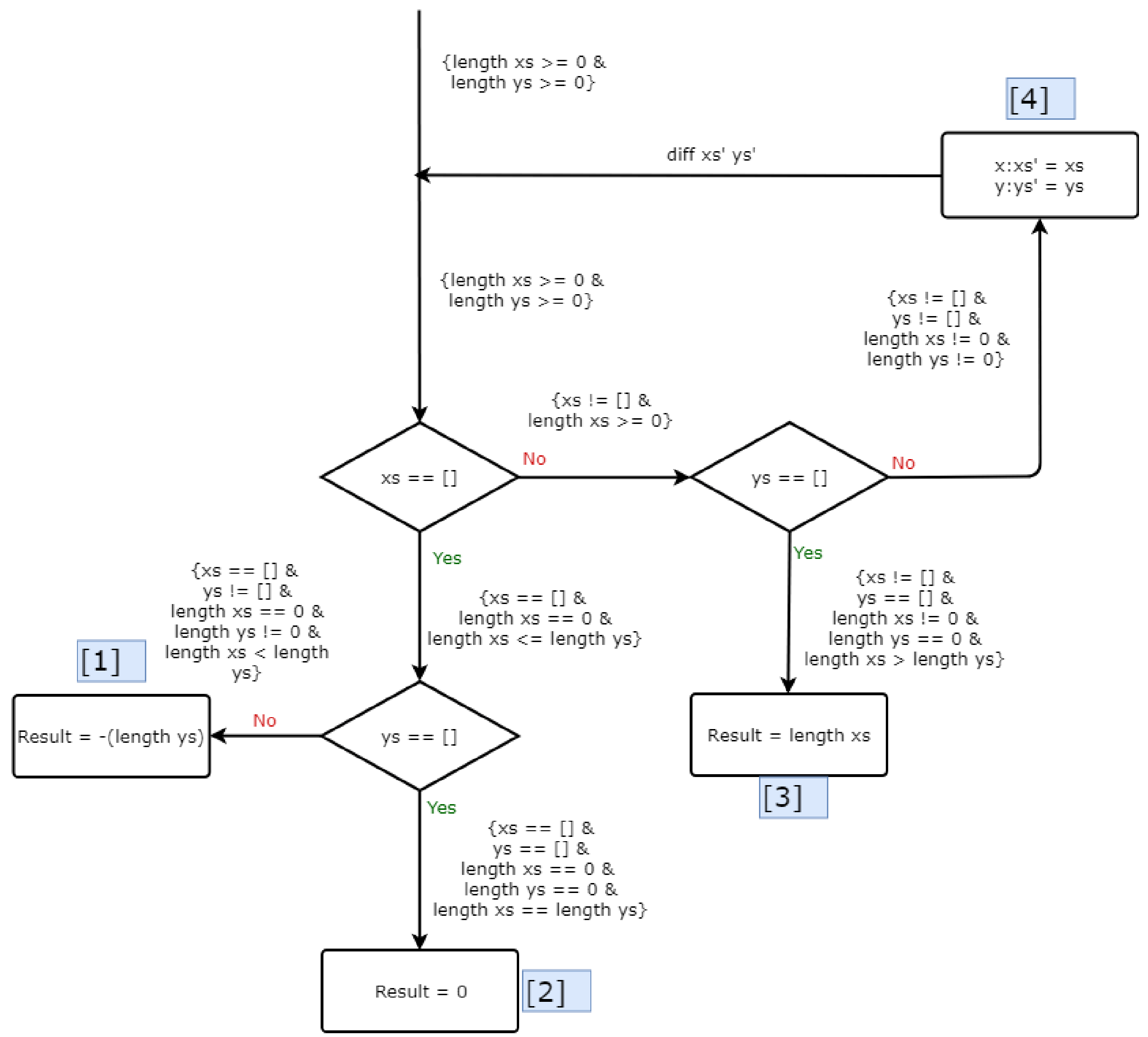

Figure 1. Flowchart (Floyd-Hoare Logic) of `diff` function

Figure 1 showed the flowchart of function `diff` drawn according to Floyd-Hoare Logic. To show that `diff` will definitely halt, four cases are considered.

Case [1]**:** *Base case.* This is when `xs` is `[]` and `ys` is some finite defined list. In this case, the function `diff` will always halt because the pattern `([], ys)` is defined in the case expressions of `diff`, and the result is `- (length ys)`. Therefore, `diff` will directly return the result whenever the pattern is matched.

Case [2]**:** *Base case.* This is when both `xs` and `ys` are `[]`. The pattern `([], [])` is defined in the case expressions of `diff` as well, and the result is `0`. Hence, `diff` will halt when `xs` and `ys` are `[]`.

Case [**3**]**:** *Base case*. This is the opposite case of **Case** [**1**], where `xs` is some finite defined list, and `ys` is `[]`. For this case, `diff` will halt as well because the pattern `(xs, [])` is also defined in the case expressions of `diff`, and the result is `length xs`.

Case [**4**]**:** *Recursive case*. When none of the patterns in **Case** [**1**], **Case** [**2**], and **Case** [**3**] are matched, then the function `diff` will call itself with arguments `xs'` and `ys'`, as shown in **Figure 1** above, where

- `xs'` is defined as the tail of the original list `xs` (`x:xs' = xs`)
- `ys'` is defined as the tail of the orignal list `ys` (`y:ys' = ys`)

For every recursion call, the function `diff` will be taking `xs'` as the new `xs`, and `ys'` as the new `ys`. Since `xs'` and `ys'` are the tail of their original list, therefore, `xs'` and `ys'` are one length shorter than their previous list. The recursion stops when the pattern of the 'new' `(xs, ys)` is matched with one of the *Base case*.

Now it is required to show that the *recursive case* of `diff` will eventually halt. It is known that

1. **Condition 1:** The `length` of a list cannot be less than `0`
2. **Condition 2:** `xs` and `ys` are some finite defined lists

The recursion terminates when it is one of the case:

1. **Case** [**1**]**:** `diff([], ys)`, which means `length xs < length ys`
2. **Case** [**2**]**:** `diff([], [])`, which means `length xs = length ys`
3. **Case** [**3**]**:** `diff(xs, [])`, which means `length xs > length ys`

Since the recursive step of `diff` is just removing the head of both lists, and **Condition 1** and **Condition 2** must be satisfied. After finite amount of recursion step, eventually one or both of the lists will become empty. As a result, it must be **Case** [**1**], **Case** [**2**] or **Case** [**3**] which causes the recursion to terminates. Therefore, it has been proved that the function `diff` will definitely halt.

□

It has been proved that

1. *Correctness*: For all finite defined lists, by giving `diff` and $\boldsymbol{\delta}$ the same input, `diff` should produced the same output as $\boldsymbol{\delta}$, and the output produced is the right answer.
2. *Total correctness*: The function `diff` will definitely halt.

Hence, the function `diff` is a totally correct implementation of the function $\boldsymbol{\delta}$.

□

Question 4.

Since the function sRev involves calling other two function, sPop and sTop. Therefore, it is required to convert sPop to sPopT and sTop to sTopT where the result of their type is Int. It is also required to convert sLen to sLenT to return the result in type Int instead of Nat.

In the conversions below, since Append is a data type, it is assumed that Append has no cost.

Converting sLen into sLenT,

```
sLenT :: Stream a -> Int
sLenT s = case s of
  None         -> 0
  Append s' _  -> 1 + (sLenT s')
```

Converting sPop into sPopT,

```
T_sPop None = 1 + T(None) -- cost:case
            = 1 + 0       -- cost:var
            = 1           -- arithmetic

T_sPop Append None _  = 1 + T(None)  -- cost:case
                      = 1 + 0        -- cost:var
                      = 1            -- arithmetic

T_sPop Append s' x  = 1 + T(Append (sPop s') x)        -- cost:case
                    = 1 + T(Append) + T(sPop s') + T(x) -- cost:func
                    = 1 + 0 + T(sPop s') + 0           -- cost:const
                    = 1 + 0 + T_sPop s' + 0            -- cost:func
                    = 1 + T(sPop s')                   -- arithmetic
                    = 1 + T_sPop s'                    -- cost:func
                    = 1 + sPopT s'                -- substitution of sPopT
                    = 1 + sLenT s' + 1            -- substitution of sLenT
                    = 2 + sLenT s'                -- arithmetic
```

Since sPop is a recursive function, where it loops all the way to the first Append (Stream a) a and remove it. Therefore, sPopT s' equals to sLenT s' + 1, where the 1 is for the sPopT None case because sLenT None returns 0.

```
sPopT :: Stream a -> Int
sPopT s = case s of
  None          -> 1              -- T_sPop None
  Append None _ -> 1              -- T_sPop Append None _
  Append s' x   -> 2 + sLenT s'   -- T_sPop Append s' x
```

Converting sTop into sTopT,

```
T_sTop None = 1 + T(special) -- cost:case
            = 1 + 0          -- cost:var
            = 1              -- arithmetic

T_sTop Append None x = 1 + T(x)   -- cost:case
                     = 1 + 0      -- cost:var
                     = 1          -- arithmetic

T_sTop Append s' _  = 1 + T(sTop s')    -- cost:case
                    = 1 + T_sTop s'     -- cost:func
                    = 1 + sTopT s'      -- substitution of sTopT
                    = 1 + sLenT s' + 1
                    = 2 + sLenT s'      -- arithmetic
```

sTop is a recursive function similar to sPop, where it loops all the way to the first Append (Stream a) a and returns the a. Hence, sTopT s' is equal to sLenT s' + 1 too.

```
sTopT :: Distinguished a => Stream a -> Int
sTopT s = case s of
  None            -> 1                -- T_sTop None
  Append None _   -> 1                -- T_sTop Append None s
  Append s' _     -> 2 + sLenT s'     -- T_sTop Append s' _
```

Converting sRev into sRevT,

```
T_sRev None = 1 + T(None) -- cost:case
            = 1 + 0       -- cost:var
            = 1           -- arithmetic

T_sRev Append None _ = 1 + T(s)   -- cost:case
                     = 1 + 0      -- cost:var
                     = 1          -- arithmetic

T_sRev _   = 1 + T(Append (sRev (sPop s)) (sTop s))         -- cost:case
           = 1 + T(Append) + T(sRev (sPop s)) + T(sTop s)   -- cost:func
           = 1 + 0 + T(sRev (sPop s)) + T(sTop s)           -- cost:const
           = 1 + 0 + T_sRev (sPop s) + T(sPop s) + T(sTop s) -- cost:func
           = 1 + 0 + T_sRev (sPop s) + T_sPop s + T_sTop s   -- cost:func
           = 1 + 0 + sRevT (sPop s) + sPopT s + sTopT s     -- cost:func
           = 1 + 0 + sRevT (sPop s) + (2 + sLenT s') + (2 + sLenT s') -- sub.
           = 5 + (2 * sLenT s') + sRevT (sPop s)             -- arithmetic
```

```
sRevT :: Distinguished a => Stream a -> Int
sRevT s = case s of
  None            -> 1
  Append None _   -> 1
  _               -> 5 + (2 * sLenT s') + sRevT (sPop s)
```

The main input is `Stream a`, so the most sensible '**size**' parameter for the cost function is the length of the given `Stream a`.

Finding the cost function of `sRev`,

From T_{sRev} `None = 1`,
hence C_{sRev} `(0) = 1`

From T_{sRev} `Append None _ = 1`,
hence C_{sRev} `(0) = 1`

From T_{sRev} `_ = 3 + (2 * sLenT s') + sRevT (sPop s)`,
hence C_{sRev} `(n+1) = 3 + (2 * n) +` C_{sRev} `(n)`

By recurrence rules,

```
f(0) = d
f(n) = f(n-1) + b * n + c
```

Converting C_{sRev} into closed form,

C_{sRev} `(0) = 1`
C_{sRev} `(n+1) = 5 + (2 * n) +` C_{sRev} `(n)`

C_{sRev} `(n) = 5 + (2 * (n-1)) +` C_{sRev} `(n-1)`
`=` C_{sRev} `(n-1) + (2 * n) + 3`

Substituting the values of $\boldsymbol{d = 1}$, $\boldsymbol{b = 2}$, $\boldsymbol{c = 3}$ into $f(n) = \frac{b}{2}n^2 + (c + \frac{b}{2})\,n + d$,
So the cost function of `sRev` is

$$C_{sRev}(n) = \frac{2}{2}n^2 + (3 + \frac{2}{2})\,n + 1$$
$$= n^2 + 4n + 1$$

Therefore, `sRev` is tractable, and its worst-case complexity class is $\boldsymbol{O(n^2)}$.