Haskell Prototype Implementation

Firstly, in order to identify the value of the similarity and the difference between two objects of type T simultaneous, a function ω is created. This function is used to generate a Haskell built-in pair `(Δ_T, Double)`, in which `Δ_T` represents the type of the difference between two objects of type `T` and the `Double` represents the type of the value of the similarity between them.

```
ω :: T -> T -> (Δ_T, Double)
ω A B = let δ = head_solution (solutions [(Unknown A B)])
            σ = fst (bounds_Delta δ)
        in (δ, σ)
```

The ω function invokes the `bounds_Delta` function to get the lower bound and the upper bound of the similarity interval of a partial solution between A and B. When two objects are completely compared, the values which are returned from this function will be identical and this value is the final value of the similarity. Therefore, by calling the `fst` function in Haskell library the value of the similarity will be derived.

Let A, B: T be objects of type $\mathsf{T} \in \mathcal{T}$. When the structures of A and B have not been analysed, "`Unknown A B`" is used to represent the initial partial solution. The corresponding similarity interval is $[0, 1]$. The lower bound value of `Unknown A B` is 0 and the upper bound value of it is 1. In fact, each partial solution, whose type is `Δ_T`, has its own lower bound value and upper bound value by applying this `bounds_Delta` function to them. The value of difference constructors of `Δ_T` will be discussed in the following section 2 and section 3 when talking through the implementation of primitive types and structured types separately.

```
bounds_Delta :: Δ_T -> (Double, Double)
bounds_Delta (Unknown A B) = (0, 1)
```

Another important function which is invoked by ω is the function `head_solution`. This `head_solution` is used to get the first path from a non-empty partial solution list.

If the list is empty a error will be presented. This function is developed based on that the partial solutions in the list are sorted, the shortest one always be the first element in this list.

```
head_solution :: [a] -> a
head_solution [] = error "No solution!"
head_solution (x:_) = x
```

The last function which is invoked by ω is the function `solutions`. It is used to get all paths or solutions from A to B. When there is no alternative path between A and B, there is no solution between them. Otherwise, when the value of the lower bound and the upper bound of the similarity interval are identical, the current solution will be the shortest path from A to B, hence it will be the first path in the result list. However, if the lower bound and upper bound of the similarity interval are not equal to each other, the corresponding path is a partial solution from A to B and this partial solution needs to be extended to get more alternative ways from the current one. Then in order to keep the shortest path as the head of the solutions list, these new partial solutions will be inserted into the rest partial solutions. Then the function `solutions` will be applied to this new solutions list until a solution is determined with the identical value of the lower bound and the upper bound of the similarity interval.

```
solutions :: [Δ_T] -> [Δ_T]
solutions [] = []
solutions (current_delta:cands) =
          if (lwb_Delta current_delta /= upb_Delta current_delta)
          then let extends = refine current_delta
                   inserts = ins_cands cands extends
               in solutions inserts
          else current_delta:(solutions cands)
```

For example, let $p_j = [s_{1j}, s_{2j}, \ldots, s_{(i-1)j}, u_{ij}]^{and}$ is the current partial solution from A to B, and other candidate solutions are in a list named `cands`. Since the lower bound and the upper bound of the corresponding similarity interval are not identical, this p_j needs to be refined to a list of new partial solutions by adding different steps in p_j. Let p_k and p_t be new partial solutions used to instead p_j with $p_k = [s_{1k}, s_{2k}, \ldots, s_{(i-1)k}, u_{ik}]^{and}$ and $p_t = [s_{1t}, s_{2t}, \ldots, s_{(i-1)t}, u_{it}]^{and}$, in which $s_{1k} = s_{1t} = s_{1j}$, $s_{2k} = s_{2t} = s_{2j}$, ..., but $s_{(i-1)k} \neq s_{(i-1)t}$. By adding these two step separately, the similarity interval of p_j will be changed. Then p_k and p_t will be inserted to the sorted `cands` lists in terms of their corresponding similarity

intervals.

Additionally, in this `solutions` function, two functions `refine` and `ins_cands` are invoked. The function `refine` is used to extend the confirmed steps list of a partial solution to another partial solutions list. Differences of different types in the type system $\mathcal{T}$ will be refined in terms of the structure of types. Therefore, details of this refine function for each type in this type system will be discussed in section 2 and section 3. However, the `refine` function is used to get a list of Δ_T by extracting a single Δ_T.

```
refine :: Δ_T -> [Δ_T]
```

Finally, the function `ins_cands` is used to insert a list of partial solutions into a sorted partial solutions list, in order to keep the shortest path always stay at the beginning of the solutions list. Let A be the existing sorted list, B be the list which will be inserted to A. The approach taken is inserting each element in B to A. The process of adding individual element to the existing sorted list is implemented by a function `ins_cand`.

```
ins_cands :: [Δ_T] -> [Δ_T] -> [Δ_T]
ins_cands [] [] = []
ins_cands [] (y:ys) = ins_cands [y] ys
ins_cands xs [] = xs
ins_cands xs (y:ys) = ins_cands (ins_cand xs y) ys
```

The `ins_cand` function is created to insert a Δ_T, called `new_delta`, to an existing sorted Δ_T list. This existing list contains a head element `current_delta` and a sub-list `cands`. The similarity interval of this `new_delta` will be compared with the similarity interval of the `current_delta`. The key point is the lower bound value of their similarity intervals. If the lower bound value of the similarity inteval of this `new_delta` is greater than the `current_delta`, this `new_delta` will be inserted as the first element in this list. Otherwise this `new_delta` will be inserted into the `cands` lists after the `current_delta`. However, in this case that the lower bound value of the similarity interval of this `new_delta` is equal or less than the corresponding value of the `current_delta`, but the upper bound value of the similarity interval of the `new_delta` is less than the lower bound value of the similarity interval of the `current_delta`, this `new_delta` will be ignored, since the `new_delta` in this case will never be selected. This `ins_cand` function is invoked by the

ins_cands function. These two functions are used to sort partial solutions and keep the shortest one at the beginning of this paths list.

```
ins_cand :: [Δ_T] -> Δ_T -> [Δ_T]
ins_cand [] x = [x]
ins_cand (current_delta:cands) new_delta =
         let lwb_cur = fst (bounds_Delta current_delta)
             upb_cur = snd (bounds_Delta current_delta)
             lwb_new = fst (bounds_Delta new_delta)
             upb_new = snd (bounds_Delta new_delta)
          in if lwb_new > lwb_cur then new_delta:(current_delta:cands)
             else ( if upb_new < lwb_cur then current_delta:cands
                    else current_delta:(ins_cand cands new_delta)
                  )
```

Therefore, when comparing two objects of general type $\mathsf{T} \in \mathcal{T}$, by applying the function ω to them, the difference and the value of the similarity will be determined. However, details, in particular the refine function and bounds_Delta function, of the implementation of applying this algorithm to concrete types in the type system $\mathcal{T}$ will be discussed in the following two sections 2 and 3.

Implementation of Primitive Types

Let A, B: P be objects of a primitive type, as the simplest data structure in the type system $\mathcal{T}$, there is only one path from A to B, and this path contains only one step. This p is type of $\Delta\_\mathtt{T}$. The selected constructor of it is $\Delta\_\mathtt{Prim\ Forward\ Backward}$.

$$A \to [p]^{or} \to B \text{ and } p = [s]^{and}$$

Hence, path p is a pair of functions, $p = (\overrightarrow{\delta_{\mathsf{P}}}\ ,\ \overleftarrow{\delta_{\mathsf{P}}})$, in particular, as the Unit type has only one value unit, when comparing two units, $p = (\mathsf{id}_{\mathsf{Unit}}\ ,\ \mathsf{id}_{\mathsf{Unit}})$.

When two objects are analysed as of primitive types, the value of the similarity are determined. This indicates that the lower bound value and upper bound value of the similarity interval are identical.

```
bounds_Delta (Δ_Prim (Id_F _) (Id_B _)) = (1, 1)
bounds_Delta (Δ_Prim (Forward (B x) (B y)) _) = (0, 0)
bounds_Delta (Δ_Prim (Forward (C x) (C y)) _) = (0, 0)
bounds_Delta (Δ_Prim (Forward (I n1 x) (I n2 y)) _) =
             (sim_I x y n1, sim_I x y n1)
bounds_Delta (Δ_Prim (Forward (R ϵ1 x) (R ϵ2 y)) _) =
             (sim_R x y ϵ1, sim_R x y ϵ1)
```

When two objects are identical, for example two units, the lower bound and upper bound value both are 1. Otherwise, when two objects are not identical, the lower bound and upper bound value of the similarity of objects which are of the boolean type and characters are 0. And the lower bound and upper bound value of the similarity whose type are the type of integers or real numbers depend on the corresponding functions.

```
--The similarity function of integers
sim_I :: Int -> Int -> Int -> Double
sim_I x y n = if (abs (x-y)) < n
              then 1 - (abs ((fromIntegral (x-y))/(fromIntegral n))) else 0
--The similarity function of real numbers
sim_R :: Double -> Double -> Double -> Double
sim_R x y ε = if (abs (x-y)) < ε then 1 - (abs ((x-y)/ε)) else 0
```

At the initial state of a comparison, the structure of each object has not been analysed, the constructor `Unkown T T` is selected. The similarity interval at this time is $[0, 1]$. Since the lower bound and the upper bound value of the similarity between them are not identical, the partial solution needs to be extended. This extension is implemented by developing a `refine` function.

```
-- refine primitive types
refine (Unknown Unit Unit) =
       [(Δ_Prim (Id_F Unit) (Id_B Unit))]
refine (Unknown (B t1) (B t2)) =
       if t1 == t2 then [(Δ_Prim (Id_F (B t1)) (Id_B (B t1)))]
       else [(Δ_Prim (Forward (B t1) (B t2)) (Backward (B t1) (B t2)))]
refine (Unknown (C x) (C y)) =
       if x == y then [(Δ_Prim (Id_F (C x)) (Id_B (C x)))]
       else [(Δ_Prim (Forward (C x) (C w2 y)) (Backward (C x) (C y)))]
refine (Unknown (I n1 x) (I n2 y)) =
       if x == y then [(Δ_Prim (Id_F (I n1 x)) (Id_B (I n1 x)))]
       else [(Δ_Prim (Forward (I n1 x) (I n2 y))
                     (Backward (I n1 x) (I n2 y)))]
refine (Unknown (R ϵ1 x) (R ϵ2 y)) =
       if (eqDouble x y) then [(Δ_Prim (Id_F (R ϵ1 x)) (Id_B (R ϵ1 x)))]
       else [(Δ_Prim (Forward (R ϵ1 x) (R ϵ1 y))
                     (Backward (R ϵ1 x) (R ϵ1 y)))]
```

Since it is often unstable to determine the equality between two real numbers in computer, the `eqDouble` function is created to define the equality of `Double` values.

```
eqDouble :: Double -> Double -> Bool
eqDouble x y = if (abs (x-y)) < 1.0e-6 then True else False
```

Implementation of Structured Types

Structured types are built on the primitive types and/or structured types. The similarity and the difference between two objects of the same structured types depend on the corresponding properties of their underlying types. In terms of such a feature, there will be more than one partial solution between two objects of structured type. The algorithm allows us to get the shortest path as the final solution. In following sections, details of the algorithm applying to structured data types will be organised in different types.

3.1 Pairs

Let A, B: $\mathsf{S} \times \mathsf{T}$ be pairs with $A = (x\ ,\ y)$ and $B = (u\ ,\ v)$. `Pair T T` is a constructor of data type `T` and it is used to represent the type pair. The type of the difference between two pairs was defined as a pair of the difference of their components. Hence, the difference between pairs is represented by the constructor Δ`_Pair` Δ`_T` Δ`_T` of data type Δ`_T`.

```
data T = ... | Pair T T | ...
data Δ_T = ... | Δ_Pair Δ_T Δ_T | ...
```

There is only one partial solution, p, from A to B. This p contains two steps s_1, s_2. The s_1 is identifying the difference between x and u, s_2 is determining the difference between y and v. Since the lower bound and the upper bound of the similarity interval of any `Unknown T T` constructor are not identical, this partial solution needs to be extended by using the `refine` function.

$$A \rightarrow [p]^{or} \rightarrow B \text{ and } p = [s_1, s_2]^{and}$$

```
refine (Unknown (Pair x y) (Pair u v)) =
       [(Δ_Pair (Unknown x u) (Unknown y v))]
refine (Δ_Pair x y) = [(Δ_Pair (head_solution (solutions (refine x)))
                               (head_solution (solutions (refine y))))]
```

A new alternative partial solution needs to be inserted in to the path list and always keep the shortest path as the first element in this list. Therefore, the value of the lower bound and the upper bound of the constructor Δ_Pair Δ_T Δ_T need to be determined.

```
bounds_Delta (Δ_Pair x y) = let lwb_Delta_x = fst (bounds_Delta x)
                                upb_Delta_x = snd (bounds_Delta x)
                                lwb_Delta_y = fst (bounds_Delta y)
                                upb_Delta_y = snd (bounds_Delta y)
                            in (((lwb_Delta_x + lwb_Delta_y)/2),
                                ((upb_Delta_y + upb_Delta_y)/2))
```

The difference between two pairs will be extended until the lower bound and the upper bound of the similarity interval are identical. The difference between two pairs will be determined.

3.2 Union Types

Let A, B: $l.\mathsf{S} \mid r.\mathsf{T}$ be objects of union types with A is either of the form $A = l(a_1)$ with a_1: S or of the form $A = r(a_2)$ with a_2: T and B is either of the form $B = l(b_1)$ with b_1: S or of the form $B = r(b_2)$ with b_2: T. As a constructor of data type `T`, it is built on the data type `UnionType`. A concrete object of union type is either built on the `Lt` side or the `Rt` side.

```
data T = ... | Union UnionType | ...
data UnionType = Lt T | Rt T
```

When the structure of A and B are confirmed, there is only one partial solution from A to B. The similarity and the difference between objects of union types depend on whether they come from the same side or not. Hence, the data type `UnionDelta` is created to represent the four cases of the difference between objects of union types. Then one constructor, Δ_Union `UnionDelta`, of data type Δ_T is built on this data type.

$$A \to [p]^{or} \to B$$

```
data Δ_T  = ... | Δ_Union UnionDelta | ...
data UnionDelta = LL Δ_T | RR Δ_T | LR T T | RL T T
```

When they come from the same side, the similarity and the difference between them depend on the corresponding properties of their underlying types. This partial solution contains only one step $p = [s_1]^{and}$. The initial partial solution is represented by using the `Unknown T T` constructor, each `T` is substituted by a union type object. In terms of the constructor of these union type objects, this partial solution needs to be extended by using the `refine` function.

```
refine (Unknown (Union (Lt x)) (Union (Lt y))) =
       [(Δ_Union (LL (Unknown x y)))]
refine (Unknown (Union (Rt x)) (Union (Rt y))) =
       [(Δ_Union (RR (Unknown x y)))]
refine (Unknown (Union (Lt x)) (Union (Rt y))) = [(Δ_Union (LR x y))]
refine (Unknown (Union (Rt x)) (Union (Lt y))) = [(Δ_Union (RL x y))]
refine (Δ_Union (LL d)) =
       [(Δ_Union (LL (head_solution (solutions (refine d)))))]
refine (Δ_Union (RR d)) =
       [(Δ_Union (RR (head_solution (solutions (refine d)))))]
```

In order to sort the partial solution list, and keep the shortest path as the first element in this list, any new partial solution whose lower bound value and upper bound value are not identical needs to be inserted into this list. The value of the lower bound and upper bound of `Δ_Union UnionDelta` is implemented by calling a function `bounds_UnionDelta`. In this function, in terms of the constructor of data type `UnionDelta`, the lower bound and upper bound of the difference between objects of union types has four cases. When the constructor is `LL Δ_T` or `RR Δ_T`, the lower bound and upper bound depend on the corresponding bounds value of `Δ_T`. When the constructor is `LR T T` or `RL T T`, their lower bound and upper bound both are 0.

```
bounds_Delta (Δ_Union x) = bounds_UnionDelta x
bounds_UnionDelta :: UnionDelta -> (Double, Double)
bounds_UnionDelta (LL d) = bounds_Delta d
bounds_UnionDelta (RR d) = bounds_Delta d
bounds_UnionDelta (LR x y) = (0, 0)
bounds_UnionDelta (RL x y) = (0, 0)
```

3.3 Sets

When comparing two objects of primitive types, these two objects are either similar (identical is an extreme case of similar) to each other or they are totally different. When comparing two objects of structured types, the properties depend on the corresponding properties of their underlying types. For example, when comparing two pairs, A and B, the first component of A will be compared with the first component of B and the second element of A will be compared with the second element of B. However, when comparing two sets, the most similar one-to-one mapping between elements in two sets need to be identified. When the difference between two sets has been determined, in order to generate a new set by applying the application operator to another set and this difference, the corresponding matching element and its change need to be confirmed. In this section, the implementation of properties of sets will be discussed.

Recalling the constructor `Set TypedSet` which is used to represent the finite set over type `T`. There is no duplicated elements allowed in a set, but there is no order restriction on elements in a set. The constructor of sets is build on a data type `TypedSet`. This data type and associated functions are used to implement the features of sets.

```
data T = ... | Set TypedSet | ...
data TypedSet = EmptySet | Add T TypedSet
```

The difference between sets is represented by the constructor Δ`_Set StepList` which indicates that the difference between two sets is a list of changing steps. Since solutions of comparison between two sets will be extended step by step, a `StepList` has a particular constructor `UnknownRest T T`. This constructor is used to represent the case of the structure of the rest part of a collection is not be recognised. In addition, there is only three steps at this moment, `Ins T`, `Del T` and `Chg` Δ`_T`. In the comparison between two sets, this constructor is used to represent the difference between two elements in these two sets. This last step constructor allows an investigation of deeper difference.

```
data Δ_T = ... | Δ_Set StepList | ...
data StepList = EmptyStepList | SL Step StepList | UnknownRest T T
              deriving (Eq, Show)
data Step = Ins T | Del T | Chg Δ_T deriving (Eq, Show)
```

The initial partial solution between two sets is represented by `Unknown T T`. Since the sets are distinguished by empty and non-empty sets, when comparing two sets, there are four cases need to be considered.

In the first case, both sets are empty. The initial partial solution is `Unknown (Set EmptySet) (Set EmptySet)`. It is needs to be extended to be another partial solution. There is no changing step between two empty sets, hence the difference between two empty sets is represented as `Δ_Set EmptyStepList`.

```
refine (Unknown (Set EmptySet) (Set EmptySet)) = [(Δ_Set EmptyStepList)]
```

In the second case, one set is a non-empty set and the other is an empty set. In this case, the initial partial solution is `Unknown (Set (Add x xs)) (Set EmptySet)`. Elements in the first set will be deleted one by one during the `refine` process until the first set is empty as well.

```
refine (Unknown (Set (Add x xs)) (Set EmptySet)) =
       let [(Δ_Set stepList)] = refine (Unknown (Set xs) (Set EmptySet))
        in [(Δ_Set (SL (Del x) stepList))]
```

The third case is the reverse one, in which the first set is empty while the second one is a non-empty set. Elements in the second set will be inserted one by one.

```
refine (Unknown (Set EmptySet) (Set (Add y ys))) =
       let [(Δ_Set stepList)] = refine (Unknown (Set EmptySet) (Set ys))
        in [(Δ_Set (SL (Ins y) stepList))]
```

In the last case, both two sets are non-empty. In this case, every two elements in the original set and the target set will be compared. Each pair leads to a path between these two sets. The shortest path will be the final solution between them. This process is implemented by invoking two functions, `more_to_more` function and `allchoices` function.

```
refine (Unknown (Set a) (Set b)) = let all_pairs = more_to_more a b
                                   in allchoices all_pairs a b
```

The more_to_more function is used to pair up every elements in the original set and the target set.

```
more_to_more :: TypedSet -> TypedSet -> [(T, T)]
more_to_more _ EmptySet = []
more_to_more EmptySet _ = []
more_to_more (Add x xs) (Add y ys) =
             app (one_to_more x (Add y ys)) (more_to_more xs (Add y ys))
one_to_more :: T -> TypedSet -> [(T, T)]
one_to_more x EmptySet = []
one_to_more x (Add y ys) = (x, y):(one_to_more x ys)
app :: [a] -> [a] -> [a]
app [] ys = ys
app (x:xs) ys = x:(app xs ys)
```

Since each pair leads a path between two sets. All partial solutions will be built by a function allchoices by invoking a choice function. The choice function is created to build a constructor Δ_Set StepList which is used to represent a partial solution between two sets. When any two elements are paired up, the rest elements in these two sets will be viewed as two objects whose structures have not been analysed.

```
allchoices :: [(T, T)] -> TypedSet -> TypedSet -> [Δ_T]
allchoices [] _ _ = []
allchoices (d:ds) a b = (choice d a b):(allchoices ds a b)
choice :: (T, T) -> TypedSet -> TypedSet -> Δ_T
choice _ EmptySet _ = error "The original set is empty."
choice _ _ EmptySet = error "The targert set is empty."
choice (x, y) a b = Δ_Set (SL (Chg (Unknown x y))
                    (UnknownRest (Set (getRest a x)) (Set (getRest b y))))
```

Although a list of partial solutions was identified, no one of them is the final solution, since the lower bound value and the upper bound value of the similarity interval are different. These values are determined by the corresponding function bound_Delta. The lower bound value and the upper bound value of the similarity interval of a Δ_Set is determined by the corresponding value of the StepList. Function bounds_StepList is created to get the lower bound value and the upper bound value of a StepList.

```
bounds_Delta (Δ_Set xs) = bounds_StepList xs
bounds_StepList :: StepList -> (Double, Double)
bounds_StepList (UnknownRest x y) = (0, 1)
bounds_StepList EmptyStepList = (0, 0)
bounds_StepList (SL x xs) = let lwb_Step_x = fst (bounds_Step x)
                                upb_Step_x = snd (bounds_Step x)
                                lwb_StepList_xs = fst (bounds_StepList xs)
                                upb_StepList_xs = snd (bounds_StepList xs)
                                weight = weight_StepList (SL x xs)
                            in (((2*lwb_Step_x+lwb_StepList_xs)/weight),
                                ((2*upb_Step_x+upb_StepList_xs)/weight))
```

```
bounds_Step :: Step -> (Double, Double)
bounds_Step (Ins x) = (0,0)
bounds_Step (Del x) = (0,0)
bounds_Step (Chg d) = bounds_Delta d
```

In terms of the logic of the algorithm, a partial solution needs to be extended until the lower bound and upper bound of the similarity interval are identical.

```
refine (Δ_Set stepList) = wrap_DeltaSet (refine_StepList stepList)
```

However, in order to extend a partial solution of the constructor `Δ_Set StepList`, a function `refine_StepList` is created to extend a list of steps to a list of such list of steps. More than one step list will be built by extending a step in a step list.

```
refine_StepList :: StepList -> [StepList]
refine_StepList EmptyStepList = []
refine_StepList (SL x xs) = if (lwb_Step x) == (upb_Step x)
                            then extend_StepList x (refine_StepList xs)
                            else combine_StepList (refine_Step x) xs
refine_StepList (UnknownRest x y) = release_StepList (refine (Unknown x y))
```

Let A, B: $\mathsf{Set}(\mathsf{T})$ be two non-empty sets over type T with a, b: T and $a \in A$ and $b \in B$. Let A', B': $\mathsf{Set}(\mathsf{T})$ be subsets of A and B with $A = a \cup A'$ and $B = b \cup B'$. The initial partial solution between them is represented by the constructor `Unknown A B`. By extending this `Unknown A B`, a list of partial solutions are obtained. One of them is `Δ_Set (SL (Chg (Unknown a b)) (UnknownRest A' B'))`. The lower bound value of this partial solution is 0 and the upper bound value of this partial solution is 1. In order to extend this partial solution to other ones, the `refine_StepList` function will be invoked. Since the lower bound value of the first step in this step list is not equal to the upper bound value

of this step, this step needs to be extended first by using a function `refine_Step`.

```
refine_Step :: Step -> [Δ_T]
refine_Step (Chg d) = refine d
refine_Step _ = error "Ins or Del step does not need to refined."
```

By applying this `refine_Step` function to the `Unknown a b`, a list of parital solutions between `a` and `b` will be generated. However, each partial solution between `a` and `b` is a part of the partial solution between `A` and `B`. Therefore, each partial solution between `a` and `b` will be combined with the rest steps of the partial solution between `A` and `B` by using a function `combine_StepList`. In this case, such that `combine_StepList (refine_Step (Chg (Unknown a b))) (UnknownRest A' B')`

```
combine_StepList :: [Δ_T] -> StepList -> [StepList]
combine_StepList [] stepList = []
combine_StepList (d:ds) stepList =
                 (SL (Chg d) stepList):(combine_StepList ds stepList)
```

In terms of the function `refine_StepList`, when the lower bound value and the upper bound value of the similarity interval of a step are identical, it indicates that these two elements are completely compared, then this step will be kept as the completed step in a partial solution. Then the rest of partial solution will be extended continuely. In particular, when extending the rest part of a partial solution, there will be one or more than one partial solutions, hence, this completed step needs to be inserted to be the first step in every partial solution. The following `extend_StepList` function is created to implement this process.

```
extend_StepList :: Step -> [StepList] -> [StepList]
extend_StepList step [] = []
extend_StepList step (x:xs) = (SL step x):(extend_StepList step xs)
```

The `refine_StepList` has three patterns which are need to be addressed. 1) A step list is non-empty has been discussed. 2) When this step list is empty, an empty list will be returned. 3) When this step list is represented by the constructor `UnknownRest T T`. Take the `UnknownRest A' B'` for example, if the step `Chg (Unknown a b)` has been completed extended, then `UnknownRest A' B'` will be refined by applying the `refine` function to `Unknown A' B'` for recognising the structure of `A'` and `B'`. In terms of the function `refine`, a list of partial solutions will be obtained, however, each partial solution

is only a part of a partial solution between A and B, hence a function release_StepList is created to extract every step list from the constructor of Δ_Set StepList.

```
release_StepList :: [Δ_T] -> [StepList]
release_StepList [] = []
release_StepList ((Δ_Set xs):rest) = xs:(release_StepList rest)
```

Finally, when all parts of a path are confirmed, every step list will be converted to a Δ_Set by using a function wrap_DeltaSet.

```
wrap_DeltaSet :: [StepList] -> [Δ_T]
wrap_DeltaSet [] = []
wrap_DeltaSet (x:xs) = (Δ_Set x):(wrap_DeltaSet xs)
```

Example 3.3.1. Implementations of Example 5.4.1 on page 79 and example 5.4.3 on page 82.

Let A, B: $\mathsf{Set}(\mathbb{R}_{0.5})$ be sets of real numbers of accuracy $\varepsilon = 0.5$ with $A = \{165.6, 208.6, 233.3, 255.3, 30$ and $B = \{82.0, 233.3, 209.0, 255.0\}$. The sR function was created to build a set of real numbers using the Haskell built-in list type.

```
sR :: Eps -> [Double] -> T
sR ε lst = Set (foldr (\x -> \s -> (add (R ε x) s)) EmptySet lst)
```

```
setA = sR 0.5 [165.6, 208.6, 233.3, 255.3, 300]
setB = sR 0.5 [82.0, 233.3, 209.0, 255.0]
delta_setA_setB = ω setA setB
> setA
{300.0, 255.3, 233.3, 208.6, 165.6}
> setB
{255.0, 209.0, 233.3, 82.0}
> delta_setA_setB
(Δ_Set
 [ chg. Δ_Prim 255.3, ((λx.x-0.3), (λx.x+0.3)), 255.0,
   chg. Δ_Prim (=233.3) ,
   chg. Δ_Prim 208.6, ((λx.x+0.4), (λx.x-0.4)), 209.0,
   del. 300.0,
   ins. 82.0,
   del. 165.6]
,0.35555555555554796)
```

■

3.4 Multisets

A multiset allows duplicated elements and there is no order restriction for its components. The difference between multisets is also built on a data type `StepList`. And the approach taken to identify the difference and the value of the similarity between two multisets and the implementation of the forward and backward application on multisets are similar to the investigation for sets. The details of the implementation of properties for multisets will be discussed in this section.

The constructor `MSet TypedMSet` is used to represent multisets, this constructor is built on the data type `TypedMSet`. The type of the difference between multisets is represented by a constructor `Δ_MSet StepList`.

```
data T = ... | MSet TypedMSet | ...
data TypedMSet = EmptyMSet | Put T TypedMSet
data Δ_T = ... | Δ_MSet StepList | ...
```

The approach taken to identify the difference and the value of the similarity between multisets is similar to the way of investigation of sets. This approach is to find all path between two multisets and get the shortest one as the final solution. Paths are extended by applying the `refine` function to each partial solution between two multisets.

Multisets are distinguished by empty and non-empty multisets, hence four cases need to be considered when talking about the initial partial solution between two multisets:

1) When both multisets are empty, there is only one partial solution in the path list and this partial solution is an empty step list.

```
refine (Unknown (MSet EmptyMSet) (MSet EmptyMSet)) =
       [(Δ_MSet EmptyStepList)]
```

2) When the original multiset is non-empty while the target one is empty, the final solution is a list of deletion of elements in the original multiset.

```
refine (Unknown (MSet (Put x xs)) (MSet EmptyMSet)) =
     let [(Δ_MSet stepList)] = refine (Unknown (MSet xs) (MSet EmptyMSet))
      in [(Δ_MSet (SL (Del x) stepList))]
```

3) When the original multiset is empty but the target one is non-empty, the final solution is a list of insertion of elements in the target multiset.

```
refine (Unknown (MSet EmptyMSet) (MSet (Put y ys))) =
     let [(Δ_MSet stepList)] = refine (Unknown (MSet EmptyMSet) (MSet ys))
      in [(Δ_MSet (SL (Ins y) stepList))]
```

4) When both multisets are non-empty. The number of partial solutions between them depends on the size of them. By pairing up every two elements between these two multisets, then each pair will lead a partial solution.

```
refine (Unknown (MSet a) (MSet b)) =
       let all_pairs_mset = more_to_more_mset a b
        in allchoices_mset all_pairs_mset a b
```

The process of pairing up elements in these two multisets is implemented by functions `more_to_more_mset` and `one_to_more`.

```
more_to_more_mset :: TypedMSet -> TypedMSet -> [(T, T)]
more_to_more_mset _ EmptyMSet = []
more_to_more_mset EmptyMSet _ = []
more_to_more_mset (Put x xs) (Put y ys) =
app (one_to_more_mset x (Put y ys)) (more_to_more_mset xs (Put y ys))
one_to_more_mset :: T -> TypedMSet -> [(T, T)]
one_to_more_mset x EmptyMSet = []
one_to_more_mset x (Put y ys) = (x, y):(one_to_more_mset x ys)
```

The `allchoices_mset` function is used to get all partial solutions between two multisets. For each pair between elements in the original and target multiset, the rest elements is organised in the constructor `UnknownRest T T`, in this case each `T` is substituted by the constructor `MSet TypedMset`. And this structure will be extended later to obtain the complete combination between elements in these two multisets.

```
allchoices_mset :: [(T, T)] -> TypedMSet -> TypedMSet -> [Δ_T]
allchoices_mset [] _ _ = []
allchoices_mset (d:ds) a b = (choice_mset d a b):(allchoices_mset ds a b)
```

```
choice_mset :: (T, T) -> TypedMSet -> TypedMSet -> Delta
choice_mset _ EmptyMSet _ = error "The original multiset is empty."
choice_mset _ _ EmptyMSet = error "The targert multiset is empty."
choice_mset (x, y) a b = Δ_MSet (SL (Chg (Unknown x y))
                (UnknownRest (MSet (getMRest a x)) (MSet (getMRest b y))))
```

Every partial solution needs to be extended unless the lower bound value and the upper bound value of the similarity interval are identical. The lower bound and the upper bound value are obtained by applying the function `bounds_Delta` to each partial solution.

```
bounds_Delta (Δ_MSet xs) = bounds_StepList xs
```

When the lower bound value and the upper bound value are not identical, the corresponding partial solution needs to be extended again. The `refine` function will be applied to the constructor `Δ_MSet StepList` to implement this process.

```
refine (Δ_MSet stepList) = wrap_DeltaMSet (refine_StepList stepList)
wrap_DeltaMSet :: [StepList] -> [Δ_T]
wrap_DeltaMSet [] = []
wrap_DeltaMSet (x:xs) = (Δ_MSet x):(wrap_DeltaMSet xs)
```

Example 3.4.1. Implementation of Example 5.5.2 on page 88 and Example 5.5.3 on page 89.

Let A, B: $\mathsf{MSet}(\mathbb{R}_{0.5})$ be two multisets of real numbers of accuracy number $\varepsilon = 0.5$ with $A = \langle 165.6, 208.6, 233.3, 255.3, 300\rangle$ and $B = \langle 209.0, 82.0, 233.3, 209.0, 255.0, 255.0\rangle$.

```
msetA = mR 0.5 [165.6, 208.6, 233.3, 255.3, 300.0]
msetB = mR 0.5 [209.0, 82.0, 233.3, 209.0, 255.0, 255.0]
delta_msetA_msetB = ω msetA msetB
```

The `mR` function is used to create a finite multiset of real numbers by applying it to a list of object of type $\mathbb{R}$.

```
mR :: Eps -> [Double] -> T
mR ε lst = MSet (foldr (\x -> \s -> (put (R ε x) s)) EmptyMSet lst)
```

The results are presented below:

```
> msetA
<300.0, 255.3, 233.3, 208.6, 165.6>
> msetB
<255.0, 255.0, 209.0, 209.0, 233.3, 82.0>
> delta_msetA_msetB
(Δ_MSet [chg. Δ_Prim 255.3, ((λx.x-0.3), (λx.x+0.3)), 255.0,
         chg. Δ_Prim (=233.3) ,
         chg. Δ_Prim 208.6, ((λx.x+0.4), (λx.x-0.4)), 209.0,
         del. 300.0,
         ins. 255.0,
         del. 165.6,
         ins. 209.0,
         ins. 82.0
        ]
,0.29090909090908473)
```

3.5 Lists

Elements in a finite list have two properties, one is their value and the other is their positions. These two properties have been considered when defining the properties of the finite lists such as the difference, the similarity and the application facilities. In this section, the implementation of these properties of finite lists will be demonstrated.

The constructor `List TypedList` which is built on the data type `TypedList` is used to represent the typed finite lists. The position of each element in a list depends on the chronological order of insertion of the corresponding element.

```
data T = ... | List TypedList | ...
data TypedList = EmptyList | Lst T TypedList
```

The difference between lists is represented by the constructor `Δ_List StepList` which is built on the data type `StepList`.

```
data Δ_T = ... | Δ_List StepList | ...
```

The initial state of comparison between two lists is represented by the constructor `Unknown T T`, since the lower bound value of such constructor is 0 while the upper bound value of it is 1, this partial solution needs to be extend by using the `refine` function. When

the initial two lists are empty, there will be only one final solution between them and this solution is an empty changing step list.

```
refine (Unknown (List EmptyList) (List EmptyList)) =
       [(Δ_List EmptyStepList)]
```

When the original list is non-empty while the target one is empty, the final solution between them will be a list of deletion steps and this is the only solution between them.

```
refine (Unknown (List (Lst x xs)) (List EmptyList)) =
     let [(Δ_List stepList)] = refine (Unknown (List xs) (List EmptyList))
      in [(Δ_List (SL (Del x) stepList))]
```

When the original list is empty, but the target one is non-empty, the final solution will be a list of insertion steps and this is the only solution between them as well.

```
refine (Unknown (List EmptyList) (List (Lst y ys))) =
     let [(Δ_List stepList)] = refine (Unknown (List EmptyList) (List ys))
      in [(Δ_List (SL (Ins y) stepList))]
```

The last case is when two lists are non-empty, the possible partial solutions need to be identified. Being the first element in these two lists, they only have three kinds of relations and these relations lead to different possible solution between these two lists. In addition, these partial solutions need to be sorted, since the shortest one needs to be kept as the first component in the solutions list. The `ins_cands` is used to sort these partial solutions.

```
refine (Unknown (List (Lst x xs)) (List (Lst y ys))) =
ins_cands [] [ (Δ_List (SL (Chg (Unknown x y))
                           (UnknownRest (List xs) (List ys))))
             , (Δ_List (SL (Del x)
                           (UnknownRest (List xs) (List (Lst y ys)))))
             , (Δ_List (SL (Ins y)
                           (UnknownRest (List (Lst x xs)) (List ys))))]
```

These partial solutions will be extended one by one until the lower bound value and the upper bound value of the similarity interval of any of them become identical. These value are derived by the function `bounds_Delta`.

```
bounds_Delta (Δ_List xs) = bounds_StepList xs
```

When the lower bound and the upper bound value of the similarity interval of a partial solution are not identical, this partial solution needs to be extended by using the `refine` function.

```
refine (Δ_List stepList) = wrap_DeltaList (refine_StepList stepList)
wrap_DeltaList :: [StepList] -> [Δ_T]
wrap_DeltaList [] = []
wrap_DeltaList (x:xs) = (Δ_List x):(wrap_DeltaList xs)
```

Example 3.5.1. The following code presents the implementation of two lists examples. The `lC` function was created to build a list using the Haskell built-in list type.

```
lC :: [Char] -> T
lC lst = List (foldr (\x -> \l -> (ins (C x) l)) EmptyList lst)
```

```
listA = lC ['a', 'b', 'c', 'a', 'b', 'c', 'd']
listB = lC ['c', 'a', 'a', 'b', 'c', 'd', 'a', 'b']
delta_listA_listB = ω listA listB
> listA
['a', 'b', 'c', 'a', 'b', 'c', 'd']
> listB
['c', 'a', 'a', 'b', 'c', 'd', 'a', 'b']
> delta_listA_listB
(Δ_List [ins. 'c',
         chg. Δ_Prim ('a') ,
         ins. 'a',
         chg. Δ_Prim ('b') ,
         chg. Δ_Prim ('c') ,
         ins. 'd',
         chg. Δ_Prim ('a') ,
         chg. Δ_Prim ('b') ,
         del. 'c',
         del. 'd'
        ]
,0.6666666666666666)
```

■

3.6 Mappings

A constructor `Mapping TypedMap` is used to represent the finite mapping type, and it is built on the data type `TypedMap`. The constructor `Env T T TypedMap` in `TypedMap` is used to represent a non-empty mapping, in which the first `T` refers the type of element in domain, the second `T` is the type of the element in the codomain. In addition, the difference

between two mappings are represented by the constructor Δ_Map StepList of the data type Δ_T, it is built on the data type StepList which indicates the difference between mappings is a changing step list.

```
data T = ... | Mapping TypedMap | ...
data TypedMap = EmptyMap | Env T T TypedMap
data Δ_T = ... | Δ_Map StepList | ...
```

Since a mapping can be constructed as a set of pairs and the properties of finite sets have been developed and implemented, a function map_to_set is created to organise elements in a mapping to a set of pairs.

```
map_to_set :: TypedMap -> TypedSet
map_to_set EmptyMap = EmptySet
map_to_set (Env x y tmap) = Add (Pair x y) (map_to_set tmap)
```

In order to compare two mappings Mapping map1 and Mapping map2, a Unknown T T constructor is used to represent the initial partial solution between them. Since the lower bound value and the upper bound value of such Unknown T T constructor is not identical, this initial partial solution needs to be extended by applying the function refine on it. And this process is implemented by applying the refine function to the corresponding sets which are converted from these two mappings. However, the result is a list of partial solutions between two sets, a function setDelta_to_mapDelta is created to get a list of partial solutions between two mappings.

```
refine (Unknown (Mapping map1) (Mapping map2)) =
       setDelta_to_mapDelta
       (refine (Unknown (Set (map_to_set map1)) (Set (map_to_set map2))))
setDelta_to_mapDelta :: [Δ_T] -> [Δ_T]
setDelta_to_mapDelta [] = []
setDelta_to_mapDelta ((Δ_Set steps):rest) =
                     (Δ_Map steps):(setDelta_to_mapDelta rest)
```

In term of the logic of the algorithm, when the lower bound value and the upper bound value of the similarity interval of any partial solution is not identical, this partial solution needs to be extended. The function bounds_Delta is used to implement this process. The following Haskell code is used to get the corresponding value of partial solution of the constructor Δ_Map StepList.

```
bounds_Delta (Δ_Map xs) = bounds_StepList xs
```

And by applying the `refine` function to the partial solution of the instance of the constructor `Δ_Map StepList`, some new partial solutions will be generated. The one with the identical lower bound and upper bound value of the similarity interval will be selected as the final solution between these two mappings.

```
refine (Δ_Map stepList) = wrap_DeltaMap (refine_StepList stepList)
wrap_DeltaMap :: [StepList] -> [Δ_T]
wrap_DeltaMap [] = []
wrap_DeltaMap (x:xs) = (Δ_Map x):(wrap_DeltaMap xs)
```

Example 3.6.1. Implementation of Example 5.6.1 on page 92 and Example 5.6.2 on page 94.

An identity function `f` and another function `g` were considered to demonstrate the similarity and the difference between two mappings in a finite domain.

```
f :: T -> T
f (R eps x) = R eps x
g :: T -> T
g (R eps x) = R eps (x-0.3)
```

Two mappings were built by applying the function `mapT` to the function and the domain of this function. And the function `sR` refers to Example 3.3.1 on page 15.

```
mapf = mapT f (sR 0.5 [165.6, 208.6, 233.3, 255.3, 300.0])
mapg = mapT g (sR 0.5 [82.0, 233.3, 209.0, 255.0])
delta_f_g = ω mapf mapg
> delta_f_g
(Δ_Map [ chg. Δ_Pair (Δ_Prim 255.3, ((λx.x-0.3), (λx.x+0.3)), 255.0)
                (Δ_Prim (255.3 => 254.7) ),
         chg. Δ_Pair (Δ_Prim (=233.3) )
                (Δ_Prim 233.3, ((λx.x-0.3), (λx.x+0.3)), 233.0),
         chg. Δ_Pair (Δ_Prim 208.6, ((λx.x+0.4), (λx.x-0.4)), 209.0)
                (Δ_Prim 208.6, ((λx.x+0.1), (λx.x-0.1)), 208.7),
         del. (300.0,300.0),
         ins. (82.0, 81.7),
         del. (165.6,165.6)
       ]
,0.31111111111110606)
```

■