Daraus kann eine Zuordnung $\overline{v_i} \mapsto \vec{v}'_i := \mathtt{bag}(\vec{v}_i)$ bestimmt werden, mit der die Eingabe $[\vec{v}_1, \ldots, \vec{v}_n]$ in $[\vec{v}'_1, \ldots, \vec{v}'_n]$ umgewandelt werden kann (in $O(n)$, wenn man sich in a) die Liste $[\overline{v_1}, \ldots, \overline{v_n}]$ merkt.)

Trennung nach *set*-Gleichheit

$$d_{\mathtt{set}(\tau)}(\tau^* \times \iota)^* \to \iota^{**}$$

(i) Definiere *weaksort* $: \tau^{**} \to \tau^{**}$ wie bei der Trennung nach *bag*-Gleichheit, nur daß im Schritt (c) aus den Listen $[w_{i,j_1}, \ldots, w_{i,j_{k_i}}]$ keine Multimenge, sondern eine (schwach sortierte) Menge gebildet wird, d.h. Duplikate ignoriert werden.

(ii) Berechne aus $[(\vec{v}_1, w_1), \ldots, (\vec{v}_n, w_n)]$ zuerst $[\vec{v}'_1, \ldots, \vec{v}'_n] = \mathit{weaksort}([\vec{v}_1, \ldots, \vec{v}_n])$ und wende den Trenner d_{τ^*} auf $[(\vec{v}'_1, w_1), \ldots, (\vec{v}'_n, w_n)]$ an. Das gibt das korrekte Ergebnis, da $\vec{v}'_i =_{\tau^*} \vec{v}'_j$ genau dann gilt, wenn $\vec{v}_i =_{\mathtt{set}(\tau)} \vec{v}_j$, weil die Komponenten in $\vec{v}'_i, \vec{v}'_j$ in schwacher Sortierung und jeweils nur einmal auftreten.

Beispiel: Schwache Sortierung für Multimengen

Sortiere `[aba,bca,aab,ba,bcb]` mit $d_{\texttt{bag(char)}}$.

(i) Paare die Elemente mit Adressen ihrer Listen und sich:

```
aba  ↦  [(a,(aba,a)),(b,(aba,b)),(a,(aba,a))]
      ⋮
bcb  ↦  [(b,(bcb,b)),(c,(bcb,b)),(b,(bcb,b))]
```

(ii) Trenne die Verkettung der erhaltenen Listen mit $d_{\texttt{char}}$:

```
[ [(aba,a),(aba,a),(bca,a),(aab,a),(aab,a),(ba,a)],
  [(aba,b),(bca,b),(aab,b),(ba,b),(bcb,b),(bcb,b)],
  [(bca,c),(bcb,c)] ]
```

(iii) Trenne deren Verkettung mit $d_{\texttt{ref(char}^*\texttt{)}}$ nach Gleichheit der String-Adressen:

```
    [ [a,a,b],[a,b,c],[a,a,b],[a,b],[b,b,c] ]
 =  [ aab,abc,aab,ab,bbc ]
```

Im Programm wird eine andere Anordnung erzeugt:

⟨*discriminate.expl.sml*⟩+≡

```
bagsort [[C #"a",C #"b",C #"a"], [C #"b",C #"c",C #"a"],
         [C #"a",C #"a",C #"b"], [C #"b",C #"a"],
         [C #"b",C #"c",C #"b"]];
```

⟨*Ergebnis*⟩+≡

```
val it =
  [[C #"b",C #"a",C #"a"],[C #"c",C #"b",C #"a"],
   [C #"b",C #"a",C #"a"],
   [C #"b",C #"a"],[C #"c",C #"b",C #"b"]] : U list list
```

Anwendung: Dokumentklassifizierung

Klassifiziere Dokumente nach darin vorkommenden Begriffen.

Vereinfacht:

Gruppiere Zeichenreihen nach darin vorkommenden Buchstaben.

⟨*discriminate.expl.sml*⟩+≡

```
fun discrChar (s : string list) =
    let
        fun string2Uref str = R (ref (S str))
        fun cPair (u as (R(Ref as (ref(S str))))) =
            map (fn c => (C c, Ref)) (String.explode str)
        val Lists = map cPair (map string2Uref s)
    in
        discriminate (List.concat Lists)
    end;

map (map toString) (discrChar  ["abc","aa","b","ac"]);
```

⟨*Ergebnis*⟩+≡

```
val it = [["abc","aa","aa","ac"],["abc","b"],["abc","ac"]] : string list list
```

Ignorieren wir die Einbettungen in den Typ `U`, so bildet man zu jeder Zeichenreihe die Paare aus den Zeichen und der Reihe,

```
abc  ↦  [(a,abc),(b,abc),(c,abc)]
 aa  ↦  [(a,aa),(a,aa)]
  b  ↦  [(b,b)]
 ac  ↦  [(a,ac),(c,ac)]
```

und trennt die Verkettung der Listen nach den Zeichen.

Anwendung: Index aller Wörter eines Texts

Der Index des Texts gibt zu den Wörtern ihre Vorkommen an.

Man paart jedes Wort mit seiner Position und trennt die Paare dann nach den Wörtern:

⟨*discriminate.expl.sml*⟩+≡
```
fun discrString text =
    let
        fun addPosition (s::ss) n =
            (S s, int_toInfo n)::(addPosition ss (n+1))
          | addPosition [] n = []
    in
        (discriminateToStrings (addPosition text 1))
    end;

discrString ["aber", "er", "aber", "so", "und", "so"];
```

⟨*Ergebnis*⟩+≡
```
val it = [["1","3"],["2"],["4","6"],["5"]] : string list list
```

Um die Wörter in der Ausgabe zu haben, paart man jedes w im Text mit seiner Position i und w selbst, und trennt die entstandene Liste der $(w, (w, i))$ nach den Wörtern.

Lokale Hilfsfunktionen

⟨*Hilfsfunktionen*⟩≡

```
local
    fun unC (C c,w) = (c,w)
      | unC _ = raise Inhomogenity

    fun reassoc [] = [] : (U * info) list
      | reassoc ((P(u,v),w) :: uvws) =
                 (u,ref(P(v,R w))) :: (reassoc uvws)
      | reassoc (_ :: uvws) = raise Inhomogenity

    fun unP (ref(P(v,R w))) = (v,w)
      | unP _ = raise Inhomogenity

    fun splitList ((L v,w)::vws) =
        let val (A,B) = splitList vws
        in case v of [] => (w::A,B)
                   | (h::r) => (A,(P(h,L r),w)::B)
        end
      | splitList (_::vws) = raise Inhomogenity
      | splitList [] = ([],[])

    fun unN (N(v,e),w) = (ref{content=v,eq=e},w)
      | unN _ = raise Inhomogenity

    fun addClass cs =
        let
             val (i,eqi) = (ref (L cs), ref ([]:class))
             fun pack c = (c, ref (P(N(i,eqi),P(R i,c))))
        in
             map pack cs
```

```
        end

    fun toRefs (ref (P (N(i,eqi), u)))
        = (N(i,eqi), ref u)
      | toRefs _ = raise Undefined

    fun collect f (ws as ((ref (P (R i,c)))::_))
        = (i, f [] ws)
      | collect f _ = raise Undefined

    fun subst ((lst,us:U list)::ps) (l:U list) =
        if (L l) = !lst then us else subst ps l
      | subst [] l = raise Undefined
  in
```

⟨*Trennung nach dem Typ des ersten Listenelements*⟩≡

```
  fun discriminate ([] : (U * info) list) = [] : class list
    | discriminate [(v,w)] = [[w]]
    | discriminate (vws as ((v,w) :: _)) =
      case v of
          (C _) => CharDiscr.discriminate (map unC vws)
        | (L _) =>
          let
              val (A,B) = splitList vws
              val Bs = discriminate B
          in
              if A = [] then Bs else A :: Bs
          end
        | P(_,_) =>
          let
              val Ss = discriminate (reassoc vws)
              val Ts = map (map unP) Ss
```

```
      in
          List.concat (map discriminate Ts)
      end
    | (S _) =>
      let
          fun toCs (S s) = L (map C (String.explode s))
            | toCs _ = raise Inhomogenity
      in
          discriminate (map (fn(v,w) => (toCs v,w)) vws)
      end
    | (N _) => (* modulo equivalence on adresses *)
      RefDiscr.discriminate (map unN vws)
    | (R _) => (* modulo identiy on adresses *)
      raise Undefined
      (* If we had defined N : U ref * U ref -> U rather than
         N : U ref * U ref list ref -> U, using L : U ref list -> U,
         we could use the references as the equivalence classes:
      (R _) =>
      let
          val us = map (fn (R v,w) => !v) vws
          fun toN (R v,w) = (v := ([] : class) ; (N(v,v),w))
            | toN _ = raise Undefined
          val ws = discriminate (map toN vws)
      in
          "restore the values assigned to vs in vws using us";
          ws
      end
      *)
    | (Bag _) =>
      let
          val (vs,ws) = ListPair.unzip vws
          fun unBag (Bag v) = v | unBag _ = raise Undefined
```

```
            val svs = map L (bagsort (map unBag vs))
            val newvws = ListPair.zip (svs,ws)
        in
            discriminate newvws
        end
      | (Set _) =>
        let
            val (vs,ws) = ListPair.unzip vws
            fun unSet (Set v) = v | unSet _ = raise Undefined
            val svs = map L (setsort (map unSet vs))
            val newvws = ListPair.zip (svs,ws)
        in
            discriminate newvws
        end

and bagsort us = (* Standard weak sorter for bags *)
    let
        val byElements =
            discriminate (List.concat (map addClass us))
        val byLists =
            discriminate (map toRefs (List.concat byElements))
        fun asBag acc [] = acc : U list
          | asBag acc ((ref (P (R i,c)))::ws)
            = asBag (c::acc) ws
          | asBag _ _ = raise Undefined
        val Subst = map (collect asBag) byLists
    in
        map (subst Subst) us
    end
```

⟨*Schwache Sortierung von Mengen*⟩≡

```
and setsort us = (* Standard weak sorter for sets *)
```

```
    let
        val byElements =
            discriminate (List.concat (map addClass us))
        val byLists =
            discriminate (map toRefs (List.concat byElements))
        fun asSet acc [] = acc : U list
          | asSet [] ((ref (P (R i,d)))::ws) =
            asSet [d] ws
          | asSet (c::acc) ((ref (P (R i,d)))::ws) =
            if d = c then asSet (c::acc) ws
               else asSet (d::c::acc) ws
          | asSet _ _ = raise Undefined
        val Subst = map (collect asSet) byLists
    in
        map (subst Subst) us
    end

 fun discriminateToStrings l =
     map (map toString) (discriminate l)
end
```