Multiset Discrimination

R.Paige, R.Tarjan e.a.

Literatur.:

J.Cai, R.Paige: Using multiset discrimination to solve language processing problems without hashing. TCS 145 (1995), 189-228

F.Henglein: Multiset Discrimination. `http://www.Plan-X.org`

Problem:

Gegeben eine Multimenge (als Liste von Objekten, bei der die Anordnung irrelevant ist). Unterscheide voneinander verschiedene Elemente und identifiziere gleiche Elemente.

Verallgemeinerung 1:

Gegeben eine Multimenge und eine Äquivalenzrelationen auf den Elementen. Unterscheide inäquivalente Elemente und fasse äquivalente in Klassen zusammen.

Verallgemeinerung 2:

Gegeben eine Multimenge von (Objekt,Info)-Paaren und eine Äquivalenzrelation auf den Objekten. Fasse die Informationen zu den Objekten derselben Äquivalenzklassen zusammen.

Ziel:

- ohne Hashing,

- Linearzeit-Algorithmus für azyklische Objekte,

- $n \log n$-Algorithmus für Graphen (z.B. Automaten)

Anwendungen

(i) Bei 2 Elementen: Gleichheits- bzw. Äquivalenztest

  (a) Isomorphietest bei Bäumen

  (b) Dagifizierung von Bäumen (Trie → DAWG)

  (c) Minimierung von det. Automaten

  (d) Termgleichheit bei ass./komm./idempot.Funktionen

(ii) Bei größeren Multimengen:

  (a) Sind zwei Listen von Ganzzahlen dieselbe Menge?

  (b) Finde Duplikate in einer Liste von Strings

  (c) Verbesserung lexikographischen Sortierens

Vorteil der Diskriminierung: in vielen Fällen

(i) kann Hashing und Sortieren ersetzen

(ii) ist asymptotisch und praktisch linear bzw. $n \log n$

(iii) ist relativ einfach zu implementieren (Zeigermaschine)

(iv) ist auch für rekursive Datentypen möglich

Trennfunktionen

Sei τ ein Typ und $\sim$ eine Äquivalenzrelation auf V_τ. Ein *Trenner* $d : \tau^* \to (\tau^*)^*$ ordnet jeder Liste $\vec{v} = [v_1, \ldots, v_n] : \tau^*$ eine Liste $d(\vec{v}) = [\vec{u}_1, \ldots, \vec{u}_k] : (\tau^*)^*$ von τ-Listen zu, so daß

(i) die Verkettung von $d(\vec{v})$ eine Permutation von $\vec{v}$ ist und

(ii) jedes $\vec{u}_i$, $1 \leq i \leq k$, eine Permutation der Teilfolge aller $v \sim v_i$ von $\vec{v}$ ist, für ein v_i in $\vec{v}$.

Der Trenner d ist *ordnungserhaltend*, wenn die Elemente in jedem u_i in derselben Reihenfolge wie in $\vec{v}$ vorkommen.

Verallgemeinerte Form mit $d_\tau : (\tau \times \iota)^* \to (\iota^*)^*$:

Eingabeliste $[(v_1, w_1), \ldots, (v_n, w_n)]$, wobei die Ausgabe bei $\vec{u}_i$ statt $\vec{u}_i = [v_{i,1}, \ldots, v_{i,k}]$ die zugehörige Liste $[w_{i,1}, \ldots, w_{i,k}] : \iota^*$ der $w_{i,j}$ mit $(v_{i,j}, w_{i,j})$ in der Eingabe hat.

Die $w_i : \iota$ sind Informationen eines festen Typs ι und gehen nicht in die Äquivalenzrelation $\sim$ ein. (Einfache Form:$w_i = v_i$)

Unterscheidungsproblem

Das *Unterscheidungsproblem* für τ modulo $\sim$ besteht darin, eine Trennfunktion d_τ zu finden, die $d_\tau(\vec{v})$ in der Zeit $O(|\vec{v}|)$ berechnet. Das wird induktiv über den Aufbau von τ und unabhängig von den w_i erfolgen.

Typen τ **und** ***Werte*** $v : \tau$

τ	$:=$		v	$:=$	
Grundtyp		a			c
Einheitstyp	$\mid$	1	$\mid$		$()$
Paare	$\mid$	$\tau_1 \times \tau_2$	$\mid$		(v_1, v_2)
Disj.Vereinigung	$\mid$	$(\tau_1 + \tau_2)$	$\mid$		$in_l(v) \mid in_r(v)$
Adresse	$\mid$	$ref(\tau)$	$\mid$		n
Typvariable	$\mid$	α			
Induktiver Typ	$\mid$	$\mu\alpha.\tau$			
a	$:=$	`char` $\mid$ `bool`.			
τ^*	$:=$	$\mu\alpha(1 + \tau \times \alpha)$			
`string`	$:=$	`char`*.			

$V_a :=$ der Bereich der Werte des Grundtyps a.

$V[N] :=$ Bereich aller aus $N \subseteq \mathbb{N}$ und den V_a gebildeten Werte.

Typisierung von Werten

Werte enthalten i.a. Adressen n. Sie werden in einem Kontext Γ von Typannahmen $n : \tau$ nach folgenden Regeln getypt:

$$\frac{}{\Gamma \vdash 1 : ()} \qquad \frac{}{\Gamma \vdash c : a}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash v : \tau_1}{\Gamma \vdash in_l(v) : (\tau_1 + \tau_2)} \qquad \frac{\Gamma \vdash v : \tau_2}{\Gamma \vdash in_r(v) : (\tau_1 + \tau_2)}$$

$$\frac{n : ref(\tau) \in \Gamma}{\Gamma \vdash n : ref(\tau)}$$

$$\frac{\Gamma \vdash v : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash v : \mu\alpha.\tau}$$

V_τ sei der Bereich der Werte vom Typ τ.

Größe von Objekten

Auf den Grundtypen a sei eine *Größe* $|\cdot|_a : V_a \to \mathbb{N}$ und eine Äquivalenzrelation $=_a$ gegeben.

Sie wird wie folgt auf beliebige Werte fortgesetzt:

$$\begin{array}{rcll} |()| & := & 0 & \\ |(v_1, v_2)| & := & |v_1| + |v_2| & \\ |in_l(v)| & := & 1 + |v| & \\ |in_r(v)| & := & 1 + |v| & \\ |v| & := & |c|_a & \text{für } v \in V_a \\ |n| & := & 1 & \end{array}$$

Beachte:

- Alle Adressen haben dieselbe Größe 1.
- Bei Paaren addiert sich die Größe der Komponenten, d.h. es wird angenommen, daß sich die Werte gleiche Teile *nicht* teilen. (Structure sharing nur via Adressen!)
- Damit Informationen $w : \iota$ die Größe 1 haben, nehmen wir für ι nicht `string`, sondern eher $ref(\texttt{string})$; wir brauchen aber, daß ι unter $\tau \times \iota$ abgeschlossen ist und nehmen daher $\iota := \bigcup_\tau ref(\tau)$,

Graphendarstellung von Objekten

Ein Wert v wird als Graph dargestellt, mit v als Marke an der Wurzel des Graphen und den Teilen an den Nachfolgerknoten; Adressen treten nur an den Blättern auf. Die Blätter *sind* keine Adressen $n \in N$, sondern nur mit Adressen markiert.

Ein „Sharing“ erfolgt noch nicht im Graphen für v, sondern erst, wenn man !-Kanten von den Blättern mit Marke n zum Graphen für $!n$ zieht.

Typisierter Speicher

Trenner $d_\tau : (\tau \times \iota)^* \to \iota^{**}$ verändern den Speicher S, d.h. sie benutzen S als implizites Argument.

Ein *Speicher* $S = (N, !)$ ist eine endliche Menge N von Adressen mit einer Auswertung $! : N \to V[N]$.

Der *in* $n \in S$ *gespeicherte* Wert ist $!n \in V[N]$. Die *Größe* des Speichers ist

$$|(N, !)| := \sum_{n \in N} |!n|.$$

Eine *Typsierung des Speichers* ist eine bezüglich $\subseteq$ maximale Menge Γ von Typannahmen $n : \mathit{ref}(\tau)$, so daß $\Gamma \vdash !n : \tau$ für alle $n : \mathit{ref}(\tau) \in \Gamma$.

Jede Äquivalenzrelation $\sim_N$ auf S *induziert*

(i) die Kongruenzrelation $\simeq_{V[N]}$ auf $V[N]$,

$$\simeq_{V[N]} := \bigcap \{ \simeq \mid \simeq \text{ ist Kongruenzrelation}, \sim_N \subseteq \simeq,\ =_a \subseteq \simeq \}$$

(ii) die Äquivalenzrelation $\sim_{(N,!)}$ auf N,

$$\sim_{(N,!)} := \{(n, n') \in N \times N \mid !n \simeq_{V[N]} !(n')\}.$$

Die durch $S = (N, !)$ definierte *Isomorphie* $\equiv_S$ ist die größte Äquivalenzrelation $\sim$ auf N, die sich selbst induziert, d.h. für die $\sim = \sim_S$ gilt.

Rekursive Definition der Trenner

Man muß i.a. den Trenner d_τ mit Hilfe einer (zu einer Typumgebung $[\alpha_i/\tau_i,\ldots]$ passenden) *Trennerumgebung* $\Gamma = [\alpha_i/d_{\tau_i},\ldots]$ definieren, die für jede freie Typvariable α_i von τ einen Trenner

$$d_{\tau_i} : \forall\beta((\tau_i \times \beta)^* \to \beta^{**}) \tag{8}$$

vorgibt. Die Trenner werden rekursiv über den Typaufbau durch *Trennerkombinatoren* D_c für die Typkonstruktoren c aus Trennern für die Teiltypen zusammengesetzt:

$$\begin{aligned}
d^\Gamma_\alpha &:= \Gamma(\alpha),\\
d^\Gamma_{\mathtt{char}} &:= D_{\mathtt{char}},\\
d^\Gamma_1 &:= D_{\mathtt{unit}},\\
d^\Gamma_{\tau\,\mathtt{ref}} &:= D_{\mathtt{ref}}(d^\Gamma_\tau),\\
d^\Gamma_{\sigma\times\tau} &:= D_\times(d^\Gamma_\sigma, d^\Gamma_\tau) =: d^\Gamma_\sigma \times d^\Gamma_\tau,\\
d^\Gamma_{\sigma+\tau} &:= D_+(d^\Gamma_\sigma, d^\Gamma_\tau) =: d^\Gamma_\sigma + d^\Gamma_\tau,\\
d^\Gamma_{\mu\alpha.\tau} &:= D_\mu(\lambda\tilde{d}.d^{\Gamma[\alpha/\tilde{d}]}_\tau) =: \mu\tilde{d}.\, d^{\Gamma[\alpha/\tilde{d}]}_\tau
\end{aligned}$$

Die einzelnen D_c folgen unten, von $D_{\mathtt{ref}}$ und D_μ nur Spezialfälle. Die Trenner für rekursive Typen sind auch rekursiv zu definieren, z.B. der Trenner für τ-Listen:

$$d^\Gamma_{\tau^*} := d^\Gamma_{\mu\alpha(1+\tau\times\alpha)} = \mu\tilde{d}.\,(d^\Gamma_1 + d^\Gamma_\tau \times \tilde{d}) =: D_*(d^\Gamma_\tau).$$

Das kann man nur in Programmiersprachen ausführen, die Typen wie in (8) erlauben (da zwei Rekursionsaufrufe eines Trenners zwei Ausgabetypen erfordern können), und wo man an die jeweilige Typumgebung herankommt.

Konstruktion von Trennern $d_\tau : (\tau \times \iota)^* \to \iota^{**}$
$\tau := \texttt{bool} \mid \texttt{char} \mid (\tau \times \tau) \mid \tau^* \mid \mathit{ref}(\tau)$.

Informationen $w : \iota$ seien oBdA stets in Adressen gehalten, also $V_\iota \subseteq N$ und $\iota := \bigcup_\tau \mathit{ref}(\tau)$.

- *Fall* $\tau = \texttt{bool}$:

  (i) Initialisiere[1] drei Adressen $n_{eq} : \mathit{ref}(\texttt{bool}^*)$, $n_t : \mathit{ref}(\iota^*)$ und $n_f : \mathit{ref}(\iota^*)$ mit leeren Listen

  $$n_{eq} := [], \quad n_t := [], \quad n_f := [].$$

  (ii) Durchlaufe die Eingabe $[(v_1, w_1), \ldots, (v_n, w_n)]$ und aktualisiere die Werte gemäß

  $$n_t := \begin{cases} w_i :: !n_t, & \text{falls } v_i = t, \\ !n_t, & \text{sonst} \end{cases}$$
  $$n_f := \begin{cases} w_i :: !n_f, & \text{falls } v_i = f, \\ !n_f, & \text{sonst} \end{cases}$$

  und falls dabei für $b : \texttt{bool}$ die Liste $!n_b$ nicht leer wird, setze $n_{eq} := b :: !n_{eq}$.

  (iii) Gib als Wert $d_{bool}([(v_1, w_1), \ldots, (v_n, w_n)])$ die Liste der $!n_b$ mit b auf der Liste $!n_{eq}$ aus, setze für diese b dann $n_b := []$ und schließlich $n_{eq} := []$.

  Das kann in $O(n)$ Schritten gemacht werden, wenn das Ablesen/Speichern eines Werts bei einer Adresse nur $O(1)$ Schritte kosten.

  Anschließend sind unter n_{eq}, n_t, n_f wieder [] gespeichert.

[1] Notation: $n := v$ meint, unter Adresse n den Wert v zu speichern.

- *Fall* `char`: Das geht analog, mit Adressen $n_{\texttt{eq}} : \mathit{ref}(\texttt{char}^*)$ und $n_b : \mathit{ref}(\iota^*)$ für jedes $b \in V_{\texttt{char}}$.

  (Das wird unten in der SML-Implementierung mit einem Array der Länge $|V_{\texttt{char}}| = 256$ gemacht.)

Implementierung

Wir bilden einen universellen Typ

$$U \quad \simeq \quad V_{\texttt{char}} + V_{\texttt{string}} + U^* + (U \times U) + \mathit{ref}(U)$$

und setzen $\iota = \mathit{ref}(U)$. Damit haben alle $w : \iota$ die Größe 1, aber man braucht lästige Einbettungen von U in ι und umgekehrt.

⟨*discriminate.sml*⟩≡

```
structure Discriminate =
  struct
    datatype U = C of char
               | S of string
               | L of U list
               | P of U * U
               | R of (U ref) (* info as value *)
               | Bag of U list
               | Set of U list
               | N of (U ref) * (U ref list ref)

    type info = U ref
    type class = info list (* infos as eq-class of nodes

    type discriminator = (U * info) list -> class list

    exception Undefined
```

Für die Trennung nach Adressen modulo $\sim_{\mathit{ref}(U)}$ wird ein Summand mit $N : \mathit{ref}(U) \times \mathit{ref}(\mathit{ref}(U)^*) \to U$ in U eingebettet; eine Trennung nach Adressengleichheit $=_{\mathit{ref}(U)}$ (ohne Angabe der Äquivalenzrelation) ist nicht implementiert, wäre aber mit einer Typänderung von N machbar (s.u.).

Beispiel: Trenner für `char`

⟨*CharDiscriminator*⟩≡

```
structure CharDiscr =
  struct
    val size = 256
    exception Size

    val dict = Array.array(size,[]:class) (* value restr
    val labels = ref [] : int list ref

    local
        fun add (v:char,w) =
            let
                val i = Char.ord v
                val ts = Array.sub(dict,i)
            in
                if List.exists (fn j => j=i) (!labels)
                then () else labels := i::(!labels);
                Array.update(dict,i,ts @ [w])
            end

        fun getLists () =
            let
                val classes = map (fn i => Array.sub(dic
                                  (rev (!labels))
            in
                List.app (fn i => Array.update(dict,i,[]
                         (!labels);
                labels := [];
                classes
            end
```

```
    in
    fun discriminate vs =
        (List.app add vs ; getLists())
        handle Subscript => raise Size
    end
  end
```

Beispiel

⟨*discriminate.expl.sml*⟩≡

```
use "discriminate.sml";
open Discriminate;

fun discriminateToStrings l =
    map (map toString) (discriminate l);

(* Example 1: Discriminate according to Char-labels *)

discriminateToStrings [(C #"a",toInfo "Affe"),
                       (C #"b",toInfo "Baer"),
                       (C #"a",toInfo "Albatross"),
                       (C #"b",int_toInfo 3)];
```

⟨*Ergebnis*⟩≡

```
val it = [["Affe","Albatross"],["Baer","3"]] : string li
```

Trenner für Paare

$$d_{\sigma\times\tau} : ((\sigma \times \tau) \times \iota)^* \to \iota^{**}$$

- *Fall* $\sigma \times \tau$:

  In der Eingabe $[(v_1, w_1), \ldots, (v_n, w_n)]$ sei $v_i = (v_{i,1}, v_{i,2})$.

  (i) Für $i = 1, \ldots, n$ speichere die zweite Komponente $v_{i,2}$ des Werts unter einer frischen Adresse $\overline{v_{i,2}}$, bilde das Paar $(\overline{v_{i,2}}, w_i) : \iota \times \iota$ und speichere diesen Wert in einer ebenfalls neuen Adresse $\overline{(\overline{v_{i,2}}, w_i)} : \iota$.

  (ii) Mit dem induktiv schon definierten Trenner d_σ trenne

  $$\begin{aligned} & d_\sigma([(v_{1,1}, \overline{(\overline{v_{1,2}}, w_1)}), \ldots, (v_{n,1}, \overline{(\overline{v_{n,2}}, w_n)})] \\ = \quad & [[\overline{(\overline{v_{1,2}}, w_1)}, \ldots], \ldots, [\overline{(\overline{v_{k,2}}, w_k)}, \ldots]] \end{aligned}$$

  nach gleichen ersten Komponenten $v_{1,i}$ der v_i.

  (iii) Ersetze die Adressen durch die dort gespeicherten Werte, und trenne jede Liste mit dem Trenner d_τ:

  $$[d_\tau([(v_{1,2}, w_1), \ldots]),\ \ldots,\ d_\tau([(v_{k,2}, w_k), \ldots])]$$

  (iv) Verkette die Ergebnislisten: das ergibt eine Trennung der w_j nach gleichen Wertpaaren $(v_{i,1}, v_{i,2})$,

  $$[[w_{1,1} \ldots, w_{1,k_1}], \ldots, [w_{r,1}, \ldots, w_{r,k_r}]].$$

  Die Speicherung in neuen Adressen ist nur wegen der Wahl des Typs ι notwendig.

Trennung nach $\sim_N$-äquivalenten Adressen

$$d_{ref(\tau)} : (ref(\tau) \times \iota)^* \to \iota^{**}$$

Die Eingabeliste $[(n_1, w_1), \ldots, (n_m, w_m)]$ soll bzgl. einer Äquivalenz $\sim_N$ auf dem Speicher N getrennt werden:

$$[[w_{1,1}, \ldots w_{1,k_1}], \ldots, [w_{r,1}, \ldots, w_{r,k_r}]]$$

wobei die $w_{i,j}$ die Informationen aller $n \sim_N n_i$ seien.

Stelle den Speicher $S = (N, !)$ und die Äquivalenz $\sim_N$ durch *disrimination records* $r = \{\texttt{cont} : \texttt{U ref}, \texttt{eq} : \texttt{N ref}\}$ dar:

Jeder Adresse n entspricht ein $r = (!n, n')$, wobei $n' \in N$ die Klasse $[n]_{\sim_N}$ darstellt, d.h. es ist

$$n_1 \sim_N n_2 \iff n_1' = n_2'$$

(aber $!n'$ muß nicht eine Liste der Elemente von $[n]_{\sim_N}$ sein).

Die Äquivalenz zweier Adressen n_1, n_2 kann dann in $O(1)$ Schritten festgestellt werden.

Änderungen von E und $\sim_N$ werden durch Änderungen bestehender $r = (!n, n')$ erreicht, und Änderungen an N durch Allozierung neuer r oder Löschung vorhandener.

Die Äquivalenz $\sim_N$ sei also unter n mit übergeben, d.h. $!n = (!n, n')$, oBdA mit $n' : ref(\iota^*)$.

Reduziere Trennung modulo $\sim_N$ auf Trennung modulo $=_N$:

$$d_{ref(\tau)}([(n_1, w_1), \ldots]) := d_{ref(\iota^*)}([(n_1', w_1), \ldots)$$

Trennung nach gleichen Adressen

$$d_{ref(\iota^*)} : (ref(\iota^*) \times \iota)^* \to \iota^{**}$$

- *Fall* $ref(\iota^*)$: Zur Eingabeliste $[(n_1, w_1), \ldots, (n_m, w_m)]$ sollen die w_i nach $=_N$-gleichen Adressen n_i getrennt werden, unabhängig von den gespeicherten Werten.

  (i) Wähle frische Adresse $n_{\mathtt{eq}} : ref(\iota^*)$, setze $n_{\mathtt{eq}} := []$ und $n_i := []$ für $i = 1, \ldots, m$.

  (ii) Für $i = 1, \ldots, m$ setze

$$n_{\mathtt{eq}} := \begin{cases} n_i :: !n_{\mathtt{eq}}, & \text{falls } !n_i = [], \\ !n_{\mathtt{eq}}, & \text{sonst} \end{cases}$$

  und anschließend (für ordnungserhaltendes $d_{\mathtt{ref}\,\iota^*}$)

$$n_i \;:= !n_i \;@\; [w_i]$$

  Danach steht unter $n_{\mathtt{eq}}$ eine Anordnung der *Menge* $\{n_1, \ldots, n_m\}$, mit jedem n_i genau einmal, und bei n_i die Liste $[w_{i,1}, \ldots, w_{i,k_i}]$ aller $w_{i,j}$, für die $(n_i, w_{i,j})$ in der Eingabeliste vorkommt.

  (iii) Gib für jedes n auf der Liste $!n_{\mathtt{eq}}$ die Liste $!n$ aus.

Achtung:

Die anfangs unter n_i gespeicherten Werte sind verloren.

Das ist harmlos, falls die n_i *nur* als Darstellung von $\sim_N$-Klassen benutzt werden: dazu braucht man nur Gleichheiten $n_i =_N n_j$, und die sind von den $E(n_i)$ unabhängig.

Voraussetzung: unter Adressen ι kann man Informationen vom Typ ι^* speichern, also $ref(\iota^*) \subseteq \iota$.

Trenner nach gleichen Adressen

$d_{ref(\tau)} : (ref(\tau) \times \iota)^* \to \iota^{**}$

- *Fall* $ref(\tau)$: Die Eingabe $[(n_1, w_1), \ldots, (n_m, w_m)]$ soll nach $=_N$-gleichen Adressen getrennt werden, wobei $n_i : ref(\tau)$.[2]

  Mit einem universellen Typ U kann man das auf den Fall $ref(\iota^* \times \tau)$ zurückführen:

  (i) Ändere die gespeicherten Werte zu $n_i := ([], !n_i)$ für $i = 1, \ldots, m$.

  (ii) Trenne $[(n_1, w_1), \ldots, (n_m, w_m)]$ wie im Fall $ref(\iota^*)$, ohne die zweiten Komponenten zu verändern.

  (iii) Setze die ursprünglichen Werte mit $n_i := second(!n_i)$ wieder her, für $i = 1, \ldots, m$.

  (iv) Gib das Ergebnis von (ii) aus.

  Da die Gleichheit der Adressen beibehalten bleibt, erhält man die korrekte Antwort. Damit im ersten Schritt kein Typfehler auftritt, muß man $ref(\tau)$ und $ref(\iota^* \times \tau)$ als Spezialfälle von $\iota = ref(U)$ behandeln können.

---

[2] Es ist zu teuer, zuerst die Gleichheiten unter den n_i zu bestimmen; das geht auch nicht mit

$$d_{\mathtt{set}(ref(\tau))}([([n_1], (n_1, w_1)), \ldots, ([n_m], (n_m, w_m))]) \\ = \quad [[(n_{j_1}, w_{j_1}), \ldots, (n_{j_{k_1}}, w_{j_{k_1}})], \ldots, [(n_{j_r}, w_{j_r}), \ldots, (n_{j_{k_r}}, w_{j_{k_r}})]],$$

denn dabei würde die schwache Sortierung der Verkettung von Einermengen $[(n_i, \widetilde{w}_i)]$ benötigt, die rekursiv die Trennung nach den n_i erfordert.

Trennung nach gleichen Adressen

⟨*Discriminator for references*⟩≡

```
structure RefDiscr =
  struct
    type discrRef = {content: U ref, eq: class ref}

    fun eqClass (v:discrRef) = #eq (!v)
    fun isEmpty (v:discrRef) = (!(eqClass v) = [])
    fun add (v:discrRef) (w:info) =
        let val c = (eqClass v) in c := (!c)@[w] en
    fun getThenResetClass (v:discrRef) =
        let val els = !(eqClass v)
        in (eqClass v) := [] ; els end

    val eqClasses = ref [] : discrRef list ref

    fun addClass (v:discrRef,w:info) =
        (if isEmpty v then
             eqClasses := v::(!eqClasses)
         else ();
         (add v w); ())

    fun discriminate (vws : (discrRef * info) list)
        let val _ = List.app addClass vws
            val classes =
                map (fn v => getThenResetClass v)
                    (!eqClasses)
        in
            eqClasses := [];  classes
        end
  end
```

Trenner für Listen

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

- *Fall* $\tau^* = \mu\alpha(1 + (\tau \times \alpha))$:

  (i) Zerlege $[(v_1, w_1), \ldots, (v_n, w_n)]$ in die Paare mit $v_i : 1$ und die mit $v_i : \tau \times \tau^*$, d.h. die mit leerer Liste bzw. nicht-leerer Liste v_i.

  (ii) Alle w_i mit $v_i : 1$ bilden eine Klasse A.

  (iii) Trenne die Teilliste der übrigen (v_i, w_i) mit $d_{\tau\times\tau^*}$ in

  $$\begin{aligned} & d_{\tau\times\tau^*}([(v_1, w_1), \ldots, (v_r, w_r)]) \\ = \quad & [[w_{1,1}, \ldots, w_{1,k_1}], \ldots, [w_{r,1}, \ldots, w_{r,k_r}]] \end{aligned}$$

  und füge A hinzu, falls A nicht leer war.

  Das ist keine Induktion über die Typausdrücke, terminiert aber, da bei der rekursiven Verwendung von d_{τ^*} nur kürzere Listen in den Eingabetupeln auftreten.

  Analog werden andere Datentypen $\mu\alpha\,\tau$ behandelt.

Trennung modulo $f : \sigma \to \tau$

Trennung einer Eingabe $[(v_1, w_1), \ldots, (v_n, w_n)]$ nicht nach gleichen Werten $v_i =_\sigma v_j$, sondern nach f-gleichen Werten,

$$v_i =^f_\sigma v_j : \Longleftrightarrow \; f(v_i) =_\tau f(v_j).$$

$$\begin{aligned} & d^f_\sigma([(v_1, w_1), \ldots, (v_n, w_n)]) \\ & \quad := \quad d_\tau([(f(v_1), w_1), \ldots, (f(v_n), w_n)]). \end{aligned}$$

Trennung nach *bag*-Gleichheit

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

Unter $[(\vec{v}_1, w_1), \ldots, (\vec{v}_n, w_n)]$ soll man diejenigen w_i in eine Liste zusammenfassen, die in einem $(\vec{v}_j, w_j)$ mit $\vec{v}_j \simeq_{\mathtt{bag}(\tau)} \vec{v}_i$ als w_j vorkommen.

1.Idee: Zuerst aus $\vec{v}$ eine neue Liste $\mathtt{bag}(\vec{v}) : \tau^*$ konstruieren, in der alle gleichen Elemente aus $\vec{v}$ nebeneinander stehen, und dann $[(\mathtt{bag}(\vec{v}_1), w_1), \ldots, (\mathtt{bag}(\vec{v}_n), w_n)]$ mit d_{τ^*} trennen.

Aber die Konstruktion von $\mathtt{bag}(\vec{v})$ erfordert schon zu viele Vergleiche, und man muß die entstandenen Listen noch nach einer Ordnung auf τ sortieren, damit die Trennung die für $= \mathtt{bag}(\tau)$ richtigen Ergebnisse liefert.

2.Effiziente Lösung (B.Paige): durch *schwache Sortierung* (erfordert keine Ordnung auf τ!)

(i) Defininiere eine Funktion $\mathit{weaksort} : \tau^{**} \to \tau^{**}$, die jede Liste $\vec{v}_i$ der Eingabe $[\vec{v}_1, \ldots, \vec{v}_n]$ in die Liste $\vec{v}'_i$ umordnet, bei der die Elemente wie ihre ersten Vorkommen in der Verkettung von $[\vec{v}_1, \ldots, \vec{v}_n]$ vorkommen.

(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 $\vec{v}_i =_{\mathtt{bag}(\tau)} \vec{v}_j$, weil die Komponenten in $\vec{v}'_i, \vec{v}'_j$ in schwacher Sortierung auftreten.

Schwache Sortierung von $[\vec{v}_1, \ldots, \vec{v}_n]$
$weaksort : \tau^{**} \to \tau^{**}$

(i) Speichere für spätere Trennung jedes $\vec{v}_i = [c_{i,1}, \ldots, c_{i,k_i}]$ in einer frischen Adresse $\overline{v_i}$ und dann für jedes j das Paar $(\overline{v_i}, c_{i,j})$ in einer frischen Adresse $w_{i,j} := \overline{(\overline{v_i}, c_{i,j})}$. Bilde für jedes i die Liste

$$\tilde{v}_i := [(c_{i,1}, (\overline{v}_i, w_{i,1})), \ldots, (c_{i,k_i}, (\overline{v_i}, w_{i,k_i}))].$$

(ii) Verkette die Ergebnisse $[\tilde{v}_1, \ldots, \tilde{v}_n]$ zu

$$[(c_{1,1}, (\overline{v_1}, w_{1,1})), \ldots, (c_{n,k_n}, (\overline{v_n}, w_{n,k_n}))]$$

und trenne diese Liste mit d_τ nach den Elementen $c_{i,j}$ (entsprechend einer bestimmten Anordnung $<_{[\vec{v}_1,\ldots,\vec{v}_n]}$ der Elemente $c_{i,j}$). Die Ergebnisliste

$$[[(\overline{v_i}, w_{i,j_1}), \ldots,], \ldots, [\ldots]]$$

faßt alle $(\overline{v_i}, w_{i,j})$ in ein Element zusammen, deren Liste v_i ein bestimmtes Element c aus der Verkettung der Argumentlisten $[\vec{v}_1, \ldots, \vec{v}_n]$ enthält.

(iii) Trenne die Verkettung dieser Ergebnisse (ordnungserhaltend) mit $d_{ref(\tau^*)}$ bezüglich der Gleichheit $=_N$ als Äquivalenz $\sim_N$: das Ergebnis

$$d_{ref(\tau^*)}([(\overline{v}_i, w_{i,j_1}), \ldots]) = [[w_{i,j_1}, \ldots, w_{i,j_{k_i}}], \ldots, [\ldots]]$$

enthält in den zweiten Komponenten der $[w_{i,j_1}, \ldots, w_{i,j_{k_i}}]$ genau die Elemente $c_{i,j}$ aus $\vec{v}_i$ in der Ordung $<_{[\vec{v}_1,\ldots,\vec{v}_n]}$ aufgeführt, in den ersten Komponenten jeweils $\overline{v_i}$.

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 $\mathit{weaksort} : \tau^{**} \to \tau^{**}$ 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:

$$\begin{array}{lcl} \texttt{aba} & \mapsto & \texttt{[(a,(aba,a)),(b,(aba,b)),(a,(aba,a))]} \\ & \vdots & \\ \texttt{bcb} & \mapsto & \texttt{[(b,(bcb,b)),(c,(bcb,b)),(b,(bcb,b))]} \end{array}$$

(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 $d_{\texttt{ref(char*)}}$ nach Gleichheit der String-Adressen:

$$\begin{array}{rl} & \texttt{[[a,a,b],[a,b,c],[a,a,b],[a,b],[b,b,c]]} \\ = & \texttt{[aab,abc,aab,ab,bbc]} \end{array}$$

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: Dokument := Zeichenreihe, Begriff := Buchstabe

⟨*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"
```

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"]]
```

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 li
    | 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)) v
  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 rat
     N : U ref * U ref list ref -> U, using L : U
     we could use the references as the equivalen
  (R _) =>
  let
      val us = map (fn (R v,w) => !v) vws
      fun toN (R v,w) = (v := ([] : class) ; (N(v
        | toN _ = raise Undefined
      val ws = discriminate (map toN vws)
  in
      "restore the values assigned to vs in vws u
      ws
  end
  *)
| (Bag _) =>
  let
      val (vs,ws) = ListPair.unzip vws
      fun unBag (Bag v) = v | unBag _ = raise Und
```

```
            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 Und
            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 byEle
        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 byEle
          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
```

Kombinatoren zum Aufbau von Trennfunktionen

Wir hatten einen universellen Typ U eigener Typen definiert, um die Trenner d_τ rekursiv über den Typaufbau von τ zu definieren. In d_τ werden ständig Typumwandlungen zwischen U und seinen Summanden durchgeführt.

Wenn man, um diese Umwandlungen zu vermeiden, die in SML eingebauten Typen verwenden will, kann man noch einen Teil der Definition von d_τ programmieren, sofern man polymorphe Rekursion hat (Emms/Leiß, SML+, TCS 1997)

Man definiert dazu *Kombinatoren*, die aus Trennern für bestimmte Typen einen Trenner für einen daraus aufgebauten Typen bilden, z.B. aus Trennern d_σ und d_τ einen Trenner $d_{\sigma\times\tau}$.

⟨*discrpr.sml*⟩≡

```
(* Hilfsfunktionen *)

fun reassoc (((u,v),w) :: rs) =
    (u,(v,w)) :: (reassoc rs)

fun concat (w::ws) = w @ (concat ws)
  | concat [] = []

fun split ((inl x,y)::vs) =
    let val (ls,rs) = split vs in ((x,y)::ls,rs) end
  | split ((inr x,y)::vs) =
    let val (ls,rs) = split vs in (ls,(x,y)::rs) end
  | split [] = ([],[])
```

⟨*discrpr.sml*⟩+≡

```
(* Kombinatoren zum Aufbau von Discriminatoren,
                 mit Fritz Henglein, 29.4.2005 *)

fun ProdDisc (da,db) xs =
    let val ys = reassoc xs
        val ss = da ys
        val ws = map db ss
    in concat ws end

fun PlusDisc (da,db) xs =
    let val (ys,zs) = split xs
        val (w1,w2) = (da ys, db zs)
    in w1 @ w2 end

fun UnitDisc (xs: (unit * 'a) list) =
    [map #2 xs]

(* Ein Rekursionskombinator ist nicht allgemein,
   aber im Einzelfall programmierbar: *)

datatype ('a,'b) sum = inl of 'a | inr of 'b
datatype 'a tree =
         Tree of (unit, 'a tree * 'a tree) sum

fun untree (Tree x,y) = (x,y)

fun treeDisc xs =
    (PlusDisc (UnitDisc,
               ProdDisc(treeDisc, treeDisc)))
        (map untree xs)
```