Unterscheidung von Multimengen

Hans Leiß
CIS, Universität München
Oettingenstr. 67, D-80538 München
leiss@cis.uni-muenchen.de

15.April 2004

Zusammenfassung

Mit „Unterscheidung von Multimengen“ sind Algorithmen von R. Paige u.a. gemeint, mit denen Listen, die verschiedene Multimengen darstellen, effizient unterschieden werden können, ohne Verwendung von Hashing und Sortierung nach vorgegebener Ordnungsrelation der Elemente.

Diese Notizen sollen F. Hengleins (vorläufige) Darstellung und Verallgemeinerung dieser Algorithmen aus [1] in einigen Punkten klären und eine Implementierung in SML angeben.

1 Das Unterscheidungsproblem

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 Listen von τ-Listen zu, sodaß

1. die Verkettung von $d(\vec{v})$ eine Permutation von $\vec{v}$ ist und
2. 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.

Wir betrachten das Problem in der leicht verallgemeinterten Form, bei der die Eingabe eine Liste $[(v_1, w_1), \ldots, (v_n, w_n)] \in (\tau \times \iota)^*$ ist und die Ausgabe in $\vec{u}_i$ statt einer Liste $[v_{i,1}, \ldots, v_{i,k}]$ die zugehörige Liste $[w_{i,1}, \ldots, w_{i,k}] \in \iota^*$ der $w_{i,j}$ mit $(v_{i,j}, w_{i,j})$ in der Eingabe ist. Die Komponenten $w_i : \iota$ sind Informationen eines festen Typs ι und gehen nicht in die Äquivalenzrelation $\sim$ ein.

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.

2 Werte und Speicher

Wir betrachten *Typen* τ und *Werte* $v : \tau$, die wie folgt aufgebaut sind:

$$
\begin{array}{rcl@{\qquad}rcl}
\text{Typ } \tau & := & & \text{Wert } v & := & \\
\text{Grundtyp} & & a & & & c \\
\text{Einheitstyp} & | & 1 & & | & () \\
\text{Paare} & | & \tau_1 \times \tau_2 & & | & (v_1, v_2) \\
\text{Disj.Vereinigung} & | & (\tau_1 + \tau_2) & & | & in_l(v) \quad | \quad in_r(v) \\
\text{Adresse} & | & ref(\tau) & & | & n \\
\text{Typvariable} & | & \alpha & & & \\
\text{Induktiver Datentyp} & | & \mu\alpha.\tau & & &
\end{array}
$$

Als Grundtypen a benutzen wir `char` und `bool`. Wir benutzen die Abkürzungen $\tau^* := \mu\alpha(1+\tau\times\alpha)$ für den Typ der Listen von Werten des Typs τ und `string` $:=$ `char`*.

Die Werte werden in einem Kontext Γ von Typannahmen $n : \tau$ nach folgenden Regeln getypt:

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

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

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

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

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

Auf den Grundtypen a sei eine *Größe* $|\cdot|_a : V_a \to \mathbb{N}$ und eine Äquivalenzrelation $=_a$ gegeben, die wie folgt auf beliebige Werte fortgesetzt wird:

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

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. Damit Informationen $w : \iota$ die Größe 1 haben, nehmen wir für ι nicht `string`, sondern eher $ref($`string`$)$; wir brauchen aber, daß ι unter $\tau \times \iota$ abgeschlossen ist und nehmen daher $\iota := \bigcup_\tau ref(\tau)$,

Ein *Speicher* $S = (N, E)$ besteht aus einer endlichen Menge N von Adressen und eine Auswertungsfunktion $E : N \to V[N]$, wobei $V[N]$ die wie oben aus N und den Werten der Grundtypen gebildeten Werte sind. (N ist also der Wertebereich $V_\iota = \bigcup_\tau V_\tau$ und $V[N]$ entspricht U.) Der *in* $n \in S$ *gespeicherte* Wert ist $E(n) \in V[N]$. Die *Größe* des Speichers ist

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

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

Jede Äquivalenzrelation $\sim_N$ auf N *induziert* die Kongruenzrelation $\simeq_{V[N]}$ auf $V[N]$, die kleinste Kongruenzrelation auf $V[N]$, die $\sim$ und alle $=_a$ umfasst, und die Äquivalenzrelation

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

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

Implementierungsmodell: 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.

Bem: 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 E-Kanten von den Blättern mit Marke n zum Graphen für $E(n)$ zieht.

3 Diskriminatoren

Zur Berechnung von Trennern $d_\tau : (\tau \times \iota)^* \to \iota^{**}$ werden speicherverändernde Funktionen betrachtet, d.h. ein implizites Argument $S = (N, E)$ benutzt. Wir nehmen an, daß die Informationen $w : \iota$ stets in Adressen gehalten werden, also oBdA $V_\iota \subseteq N$ ist.

Im Folgenden wird nur die Konstruktion von d_τ für

$$\tau := \texttt{bool} \mid \texttt{char} \mid (\tau \times \tau) \mid \tau^* \mid \mathit{ref}(\tau).$$

beschrieben, und $\iota := \bigcup_\tau \mathit{ref}(\tau)$ benutzt.

- *Fall* `bool`:
  1. Initialisiere[1] drei Adressen $n_{eq} : \mathit{ref}(\texttt{bool}^*)$, $n_t : \mathit{ref}(\iota^*)$ und $n_f : \mathit{ref}(\iota^*)$ mit leeren Listen
  $$E(n_{eq}) := [], \quad E(n_t) := [], \quad E(n_f) := [].$$
  2. Durchlaufe die Eingabe $[(v_1, w_1), \ldots, (v_n, w_n)]$ und aktualisiere die Werte gemäß
  $$E(n_t) := \begin{cases} w_i :: E(n_t), & \text{falls } v_i = t, \\ E(n_t), & \text{sonst} \end{cases} \qquad E(n_f) := \begin{cases} w_i :: E(n_f), & \text{falls } v_i = f, \\ E(n_f), & \text{sonst} \end{cases}$$
  und falls dabei für $b : \texttt{bool}$ die Liste $E(n_b)$ nicht leer wird, setze $E(n_{eq}) := b :: E(n_{eq})$.
  3. Gib als Wert $d_{bool}([(v_1, w_1), \ldots, (v_n, w_n)])$ die Liste der $E(n_b)$ mit b auf der Liste $E(n_{eq})$ aus, setze für diese b dann $E(n_b) := []$ und schließlich auch $E(n_{eq}) := []$.

  Man überlegt sich leicht, daß das in $O(n)$ Schritten gemacht werden kann, wenn das Ablesen und Speichern eines Werts bei einer Adresse nur $O(1)$ Schritte kosten. Beachte, daß anschließend unter den Adressen n_{eq}, n_t, n_f wieder leeren Listen gespeichert sind.
- *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.)
- *Fall* $\sigma \times \tau$: In der Eingabe $[(v_1, w_1), \ldots, (v_n, w_n)]$ sei $v_i = (v_{i,1}, v_{i,2})$ für $i = 1, \ldots, n$.

[1] Die Terminologie ist inkonsequent: anfangs schreibe ich $E(n)$ für den bei n gespeicherten Wert, später $!n$, und manchmal $n : \tau$ statt $n : \mathit{ref}(\tau)$. Das kommt vom Wechsel zur SML-Implementierung.

1. Für jedes $i = 1, \ldots, n$ 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 $\overline{(\overline{v_{i,2}}, w_i)} : \iota$.
2. Mit dem induktiv schon bekannten Trenner d_σ trenne

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

in Listen, die zu gleichen ersten Komponenten $v_{1,i}$ der v_i gehören.
3. Verkette die Listen, ersetze die Adressen durch die dort gespeicherten Werte, und trenne mit dem induktiv bekannten d_τ:

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

4. Das Ergebnis ist eine Trennung der w_j nach gleichen Wertpaaren $(v_{i,1}, v_{i,2})$.

- *Fall* $\mathit{ref}(\iota^* \times \texttt{bool})$: 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.

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

  $$n_{\texttt{eq}} := \begin{cases} n_i :: (!n_{\texttt{eq}}), & \text{falls } !n_i = (\_, f), \\ n_{\texttt{eq}}, & \text{sonst} \end{cases} \quad \text{und dann} \quad n_i := (w_i :: \mathit{fst}(!n_i), t)$$

  oder, für eine ordnungserhaltende Trennung, $n_i := (\mathit{fst}(!n_i)@[w_i], t)$.

  Danach ist bei $n_{\texttt{eq}}$ eine Anordnung der *Menge* $\{n_1, \ldots, n_m\}$ gespeichert, also jedes 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. (Die `bool`-Komponente braucht man nicht, wenn man statt $!n_i = (\_, f)$ auf $!n_i = []$ testet. Falls das dereferenzieren von n_i in den Tests zu teuer ist, muß man die Listenkomponente in ein weiteres `ref` verpacken.)
  3. Gib für jedes n auf der Liste $!n_{\texttt{eq}}$ die Liste $\mathit{fst}(!n)$ aus.

  Beachte, daß die unter n_i gespeicherten Werte verdorben werden. (Wenn die n_i *nur* als Repräsentanten von Klassen einer Äquivalenzrelation $\sim_N$ benutzt werden, ist das unproblematisch: dann werden nur Gleichungen $n_i =_N n_j$ ausgewertet, aber weder Dereferenzierungen $!n_i$ noch Zuweisungen $n_i := v$.)

  Wir setzen voraus, daß man unter Adressen ι auch Informationen vom Typ $\iota^* \times \texttt{bool}$ und ι^* speichern kann, also $\mathit{ref}(\iota^*) \cup \mathit{ref}(\iota^* \times \texttt{bool}) \subseteq \iota$.

- *Fall* $\mathit{ref}(\tau)$: Die Eingabe $[(n_1, w_1), \ldots, (n_m, w_m)]$ soll nach $=_N$-gleichen Adressen getrennt werden.[2] Falls man einen universellen Typ U hat, kann man das auf den Fall $\mathit{ref}(\iota^* \times \texttt{bool} \times \tau)$ zurückführen:

  1. Ändere die gespeicherten Werte zu $n_i := ([], f, !n_i)$ für $i = 1, \ldots, m$.
  2. Trenne $[(n_1, w_1), \ldots, (n_m, w_m)]$ wie im Fall $\mathit{ref}(\iota^* \times \texttt{bool})$, ohne die dritten Komponenten zu verändern.

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

$$\begin{aligned} & d_{\texttt{set}(\mathit{ref}(\tau))}([([n_1], (n_1, w_1)), \ldots, ([n_m], (n_m, w_m))]) \\ = \; & [[(n_{j_1}, w_{j_1}), \ldots, (n_{j_{k_1}}, w_{j_{k_1}})], \ldots, [(n_{j_r}, w_{j_r}), \ldots, (n_{j_{k_r}}, w_{j_{k_r}})]], \end{aligned}$$

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.

3. Setze die ursprünglichen Werte mit $n_i := \mathit{third}(!n_i)$ wieder her, für $i = 1, \dots, m$.

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

- *Fall* $\tilde{\mathit{ref}}(\tau)$:[3] Die Eingabe $[(n_1, w_1), \dots, (n_m, w_m)]$ soll nicht nach gleichen, sondern nach $\sim$-äquivalenten Adressen getrennt werden.

  Wenn man $\sim_N$ als separates Argument $f : \mathit{ref}(\tau) \to \mathit{ref}(\iota^* \times \mathtt{bool})$ an $d_{\mathit{ref}(\tau)}$ übergibt, könnte man $d'_{\mathit{ref}(\tau)} : (\mathit{ref}(\tau) \times \iota)^* \times (\iota \to \iota) \to \iota^{**}$ durch

  $$d'_{\mathit{ref}(\tau)}([(n_1, w_1), \dots, (n_m, w_m)], f) = d_{\mathit{ref}(\iota^* \times \mathtt{bool})}([(f(n_1), w_1), \dots, (f(n_m), w_m)])$$

  definieren. Damit das in $O(m)$ Schritten geht, muß $f(n)$ in $O(1)$ Schritten auswertbar sein.

  Wir setzen statt einem separaten Funktionsargument f voraus, daß die Äquivalenzklasse $[n]_\sim$ von n bezüglich $\sim$ in $E(n) = (v, n') : \tau \times \iota$ als Adresse n' mitkodiert ist, d.h. $n \sim m$ genau dann gilt, wenn $n' = m'$ an n und m (via E in $O(1)$ Schritten) abgelesen werden kann; es ist nicht notwendig, daß $E(n')$ eine Liste der zu n äquivalenten Adressen enthält.

  Dann kann man – sozusagen mit $n' = \mathit{snd}(!n)$ –

  $$d_{\mathit{ref}(\tau)}([(n_1, w_1), \dots, (n_m, w_m)]) := d_{\mathit{ref}(\iota^*)}([(n'_1, w_1), \dots, (n'_m, w_m)])$$

  setzen. Aber ist das erlaubt? Man behandelt dann ja statt $n : \mathit{ref}(\tau)$ nur den Fall $n : \mathit{ref}(\tau \times \mathit{ref}(\iota^*))$ bzw. $n : \mathit{ref}(\iota^* \times \mathtt{bool})$ wie oben.

  Was soll erreicht werden? Trennung nach der von $\sim_N$ induzierten Kongruenzrelation $\simeq_V$ oder bezüglich der von $\sim_N$ induzierten Äquivalenzrelation $\simeq_{(N,E)}$ auf N?

- *Fall* τ^**, d.h.* $\mu\alpha(1 + (\tau \times \alpha))$ (analog für andere induktive Datentypen $\mu\alpha\,\tau$):

  1. Zerlege $[(v_1, w_1), \dots, (v_n, w_n)]$ 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.
  2. Alle w_i mit $v_i : 1$ bilden eine Klasse A.
  3. Trenne die Teilliste der übrigen (v_i, w_i) mit $d_{\tau \times \tau^*}$ in (etwa)

     $$d_{\tau \times \tau^*}([(v_1, w_1), \dots, (v_r, w_r)]) = [[w_{1,1}, \dots, w_{1,k_1}], \dots, [w_{r,1}, \dots, w_{r,k_r}]]$$

     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. Im allgemeinen Fall muß man für $(\mu\alpha.\tau)(\vec{\beta})$ eine von einer Belegung $[\vec{\beta}/d_{\vec{\beta}}]$ abhängige Trennfunktion rekursiv definieren,

  $$d_{\mu\alpha.\tau}[\vec{\beta}/d_{\vec{\beta}}](\vec{v}) := (\mu d. d_\tau[\vec{\beta}/d_{\vec{\beta}}, \alpha/d])(\vec{v}).$$

Varianten:

- Trennung modulo $f : \sigma \to \tau$, d.h. Trennung einer Eingabe $[(v_1, w_1), \dots, (v_n, w_n)]$ nicht nach gleichen Werten $v_i =_\sigma v_j$, sondern nach f-gleichen Werten, $v_i =^f_\sigma v_j :\Longleftrightarrow f(v_i) =_\tau f(v_j)$:

  $$d^f_\sigma([(v_1, w_1), \dots, (v_n, w_n)]) := d_\tau([(f(v_1), w_1), \dots, (f(v_n), w_n)]).$$

[3] Das diente nur dazu, zu verstehen, was Henglein meinte. Das ist oben im Fall $\mathit{ref}(\iota^* \times \mathtt{bool})$ gemacht.

- Trennung nach *bag*-Gleichheit: $d_{\texttt{bag}(\tau)} : (\tau^* \times \iota)^* \to \iota^{**}$ soll unter $[(\vec{v}_1, w_1), \ldots, (\vec{v}_n, w_n)]$ diejenigen w_i in eine Liste zusammenfassen, die in einem $(\vec{v}_j, w_j)$ mit $\vec{v}_j \simeq_{\texttt{bag}(\tau)} \vec{v}_i$ als w_j vorkommen.[4]

  1. Defininiere *weaksort* : $\tau^{**} \to \tau^{**}$, eine *schwache Sortierung* (ohne Ordnung auf τ!), 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.
     (a) Speichere für eine 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
     $$\tilde{v}_i := [(c_{i,1}, (\overline{v_i}, w_{i,1})), \ldots, (c_{i,k_i}, (\overline{v_i}, w_{i,k_i}))].$$
     (b) 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.
     (c) 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 $\overline{v_i} \mapsto \vec{v}'_i := \texttt{bag}(\vec{v}_i)$ 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.)
  2. Berechne aus $[(\vec{v}_1, w_1), \ldots, (\vec{v}_n, w_n)]$ zuerst $[\vec{v}'_1, \ldots, \vec{v}'_n] = \textit{weaksort}([\vec{v}_1, \ldots, \vec{v}_n])$ 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 $\vec{v}_i =_{\texttt{bag}(\tau)} \vec{v}_j$, weil die Komponenten in $\vec{v}'_i, \vec{v}'_j$ in schwacher Sortierung auftreten.

- Trennung nach *set*-Gleichheit: $d_{\texttt{set}(\tau)} : (\tau^* \times \iota)^* \to \iota^{**}$

  1. 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.
  2. Berechne aus $[(\vec{v}_1, w_1), \ldots, (\vec{v}_n, w_n)]$ zuerst $[\vec{v}'_1, \ldots, \vec{v}'_n] = \textit{weaksort}([\vec{v}_1, \ldots, \vec{v}_n])$ 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 $\vec{v}_i =_{\texttt{set}(\tau)} \vec{v}_j$, weil die Komponenten in $\vec{v}'_i, \vec{v}'_j$ in schwacher Sortierung und jeweils nur einmal auftreten.

[4] Es liegt nahe, zuerst aus $\vec{v}$ eine neue Liste $\texttt{bag}(\vec{v}) : \tau^*$ bzw. $\texttt{bag}(\vec{v}) : (\tau \times \texttt{int})^*$ zu konstruieren, in der alle gleichen Elemente aus $\vec{v}$ nebeneinander stehen bzw. Elemente mit ihrer Vielfachheit angegeben werden, und dann $[(\texttt{bag}(\vec{v}_1), w_1), \ldots, (\texttt{bag}(\vec{v}_n), w_n)]$ mit d_{τ^*} bzw. $d_{(\tau \times \texttt{int})^*}$ zu trennen. Aber die Konstruktion von $\texttt{bag}(\vec{v})$ erfordert schon zu viele Vergleiche ($O(|\vec{v}| \cdot \log |\vec{v}|)$?) und außerdem muß man die entstandenen Listen erst noch nach einer Ordnung auf τ sortieren, damit die Trennung die für $=_{\texttt{bag}(\tau)}$ h richtigen Ergebnisse liefert.

4 Implementierung

4.1 Das Programm

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.

Für die Trennung nach Adressen modulo einer Äquivalenzrelation $\sim_{\mathit{ref}(U)}$ wird ein zusätzlicher 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 (siehe Kommentar im Programm).

⟨*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 associated to an eq-class of nodes *)

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

      exception Undefined

      fun toString (ref (S s)) = s
        | toString _ = raise Undefined
      and toInfo s = ref (S s)
      and int_toInfo n = toInfo (Int.toString n)

      (* Discriminator for atomic labels of type char *)

      structure CharDiscr =
        struct
          val size = 256
          exception Size

          val dict = Array.array(size,[]:class) (* escape 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(dict,i))
                                  (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

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

(* Type induction *)
```

```
exception Inhomogenity

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
  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, using disrc.ref(U list) *)
            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

        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

    end
```

4.2 Testbeispiele

Und hier ein paar Testbeispiele (ggf. `Control.Print.printDepth := 30;` benutzen):

⟨*discriminate.expl.sml*⟩≡

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

(* 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 list list
```

⟨*discriminate.expl.sml*⟩+≡
```
(* Example 2: Discriminate according to Pair-labels *)

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

⟨*Ergebnis*⟩+≡
```
val it = [["Affe"],["Albatross"],["Baer","3"]] : string list list
```

⟨*discriminate.expl.sml*⟩+≡
```
(* Example 3: Discriminate according to List-labels *)

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

⟨*Ergebnis*⟩+≡
```
val it = [["a","b"],["Affe"],["Albatross"],["Baer","3"]] : string list list
```

⟨*discriminate.expl.sml*⟩+≡
```
(* Example 4: Discriminate according to String-labels *)

discriminateToStrings
     [(S "abc",toInfo "a"), (S "ab", toInfo "b"),
      (S "abc",toInfo "c"), (S "abcd", toInfo "d"),
      (S "ab",toInfo "a"), (S "abcd", toInfo "ad")];
```

⟨*Ergebnis*⟩+≡
```
val it = [["b","a"],["a","c"],["d","ad"]] : string list list
```

⟨*discriminate.expl.sml*⟩+≡
```
(* Example 5: Discriminate according to Node-labels; values ui do
   not play a role, only equivalence classes ei *)

let
    val e1 = ref [] : info list ref
    val e2 = ref [] : info list ref
    val u1 = ref (C #"a")
    val u2 = ref (C #"b")
    val u3 = ref (L [C #"a"])
in
    discriminateToStrings
        [ (N(u1,e1),toInfo "a"), (N(u2,e2),toInfo "d"),
          (N(u3,e1),toInfo "b"), (N(u1,e2),toInfo "e") ]
end;
```

⟨*Ergebnis*⟩+≡

```
val it = [["d","e"],["a","b"]] : string list list
```

⟨*discriminate.expl.sml*⟩+≡

```
(* Example 6: Produce a mapping contains:(char -> string ref list) such
   that contains(c) selects all (refs to) strings containing c from an
   input list s of strings *)

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
```

⟨*discriminate.expl.sml*⟩+≡

```
(* Example 7: discriminate words of a text into occurrence positions *)

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
```

⟨*discriminate.expl.sml*⟩+≡

```
(* Example 8a: Weak sorter for bags *)

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
```

⟨*discriminate.expl.sml*⟩+≡
```
(* Example 8b: Weak sorter for sets *)
setsort [[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 #"c",C #"b",C #"a"],[C #"b",C #"a"],[C #"b",C #"a"],
   [C #"c",C #"b"]] : U list list
```

⟨*discriminate.expl.sml*⟩+≡
```
(* Example 9: Discriminate according to Bag-labels *)

discriminateToStrings
    [(Bag [C #"a",C #"b",C #"a"], toInfo "A"),
     (Bag [C #"b",C #"b",C #"a"], toInfo "B"),
     (Bag [C #"a",C #"a",C #"b"], toInfo "A"),
     (Bag [C #"b",C #"a",C #"a"], toInfo "A"),
     (Bag [C #"b",C #"a",C #"b"], toInfo "B")];
```

⟨*Ergebnis*⟩+≡
```
val it = [["A","A","A"],["B","B"]] : string list list
```

⟨*discriminate.expl.sml*⟩+≡
```
(* Example 10: Discriminate according to Set-labels *)

discriminateToStrings
    [(Set [C #"a",C #"b",C #"a"], toInfo "aba"),
     (Set [C #"b",C #"c",C #"a"], toInfo "bca"),
     (Set [C #"a",C #"a",C #"b"], toInfo "aab"),
     (Set [C #"b",C #"a"], toInfo "ba"),
     (Set [C #"b",C #"c",C #"b"], toInfo "bcb")];
```

⟨*Ergebnis*⟩+≡
```
val it = [["aba","aab","ba"],["bcb"],["bca"]] : string list list
```

⟨*discriminate.expl.sml*⟩+≡

```
(* Example 11: Weak sorter for bags (Henglein, p.19) *)

    fun wsort us =
        let
            fun f cs =
                let
                    val i = ref (L cs)
                    fun pair c = (c, ref (P (R i, R (ref (P (R i,c))))))
                in
                    map pair cs
                end

            val I = map f us
            val CI = List.concat I
            val Perm = List.concat (discriminate CI)
            val deref = map (op !) Perm
            val deP = map (fn (P(R a,R b)) => (!a,b)) deref
            val DdeP = discriminate deP
            fun unPack (ref (P (R r,b))) = b
        in
            map (map unPack) DdeP
        end
    and stringsToUs ss = map ((map C) o String.explode) ss;

(* Note that this is actually NOT a weak sorter, as it not only bag-sorts
   the elements of the input list, but also the input list itself. *)

wsort (stringsToUs ["topside", "lame", "male", "deposit", "one"]);
```

⟨*Ergebnis*⟩+≡

```
val it =
  [[C #"t",C #"o",C #"p",C #"s",C #"i",C #"d",C #"e"],
   [C #"t",C #"o",C #"p",C #"s",C #"i",C #"d",C #"e"],[C #"o",C #"e",C #"n"],
   [C #"e",C #"l",C #"a",C #"m"],[C #"e",C #"l",C #"a",C #"m"]] : U list list
```

Literatur

[1] Fritz Henglein. Multiset Discrimination. DIKU, University of Copenhagen. September 16th, 2003.