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 $\rightarrow$ 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	$:=$	$\mathtt{char} \mid \mathtt{bool}.$		
τ^*	$:=$	$\mu\alpha(1 + \tau \times \alpha)$		
`string`	$:=$	$\mathtt{char}^*.$		

$V_a :=$ 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 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. (Structure sharing nur via Adressen!)
- 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 $\iota := \bigcup_\tau \mathit{ref}(\tau)$,

Typisierter Speicher

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

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

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 $n : \mathit{ref}(\tau)$, so daß $\Gamma \vdash E(n) : \tau$ 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 $\sim_{(N,E)}$ auf N,

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

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.

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

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

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

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

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

  (iii) 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 $E(n_{eq}) := []$.

  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: manchmal $!n$ statt $E(n)$, manchmal $n : \tau$ statt $n : \mathit{ref}(\tau)$.

- *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

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

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

Beispiel: Trenner für `char`

$\langle$*CharDiscriminator*$\rangle\equiv$

```
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(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
```

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 list list
```