Multiset discrimination for acyclic data

Fritz Henglein
DIKU, University of Copenhagen
henglein@diku.dk

WG2.8 Worksthop, Kalvi, 2005/10/01-04

Overview

- Discrimination: Partitioning input into equivalence classes
- Basics: Types, equivalence classes, discriminators
- Top-down MSD for unshared data
- Bottom-up MSD for shared data (briefly!)
- Discussion

Multiset discrimination: The problem

- Partition a sequence of inputs into equivalence classes according to a given equivalence relation
- Examples:
  - Same word occurrences in text
  - Anagram classes of dictionary
  - Equal terms or (sub)trees
  - Equivalent states of finite state automaton
  - Bisimulation classes of labeled transition system
- Note: Generalization of equality/equivalence to from 2 to *n* arguments.

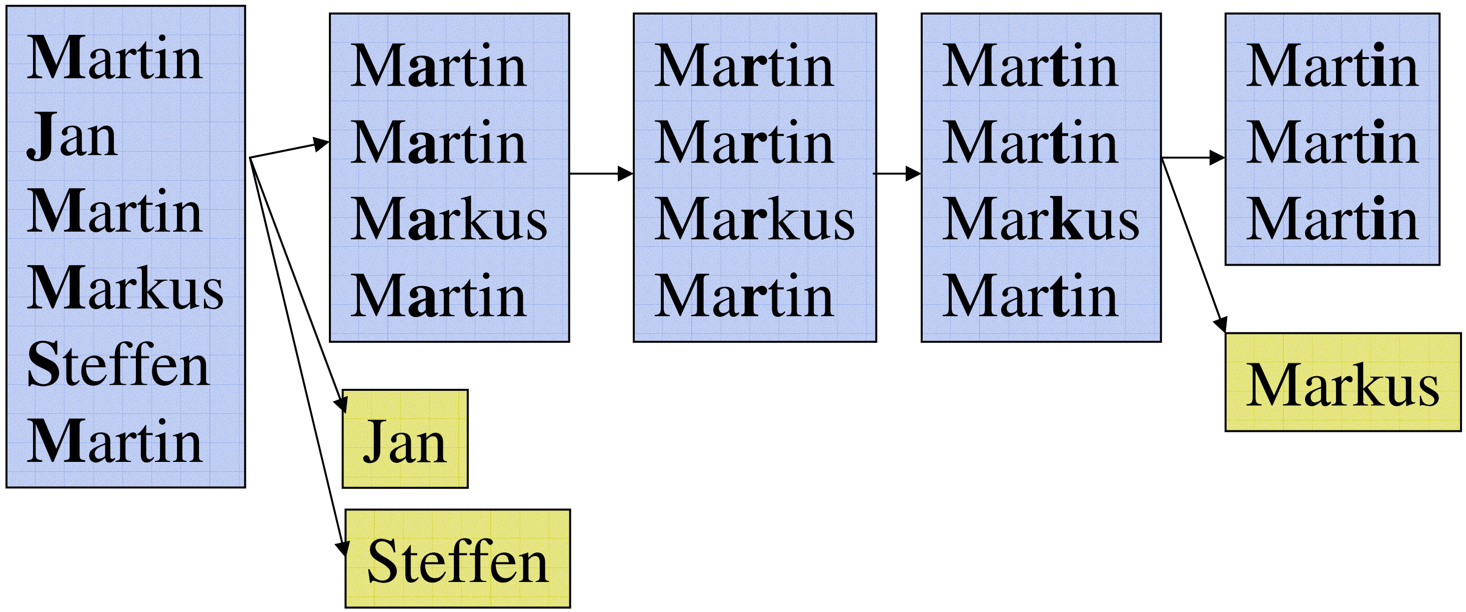
Multiset discrimination: The problem...

- Occurs frequently as auxiliary or key step in other problems; e.g.,
  - Compiling:
    - Symbol table management
    - Is there a duplicate identifier in a formal parameter list?
  - Optimization: Replace multiple equivalent data structures by (pointers to) a single data structure
- Is frequently solved by use of hashing, possibly in connection with sorting

Multiset discrimination: The techniques

- Worst-case optimal techniques for multiset discrimination without hashing or sorting
- Basic idea (for string discrimination): Partition multiset of strings according to first character, then refine blocks according to second character and so on

MSD: Basic idea



Basics: Values

- Universe *U* of first-order values:
  - $v ::= () \mid a \mid inl(v) \mid inr(v) \mid (v, v)$
  - $a ::=$ <atomic values from finite set, e.g., characters>
- Examples of values:
  (‘a’, ‘b’), inl(‘J’, inl(‘a’, inl(‘n’, inr()))))
- Notation: The latter value is also denoted by *[‘J’, ‘a’, ‘n’]* and *“Jan”.*
- Sizes of values (bit size of untyped representation):
  $|(v,v')| = |v| + |v'|$
  $|inl(v)| = |inr(v)| = 1 + |v|$
  $|()| = 0|$
  $|a| = O(\log_2 |A|)$, where $a \varepsilon A$

Basics: Types

- **Type**:
  A partial equivalence relation (per) on U; that is, a subset S of U together with an equivalence relation on S
- **Type expressions**:
  - $T ::= 1 \mid T * T \mid T + T \mid A \mid t \mid \mu t.T \mid$
    $\mid Bag(T) \mid Set(T)$
  - A ::= <atomic type names, e.g., Char>
- **Abbreviations**: $Seq(T) = \mu t.\ 1 + T * t$
  $String = Seq(Char)$
  $Bool = 1+1$

Basics: Types...

- Each type expression denotes a type:
  - *A*: primitive values with built-in equality (e.g., characters with character equality)
  - *1*: *{ () }* with *() = ()*
  - *T * T'*: *{ (t, t'): t ε T, t' ε T' }* with canonically induced equivalence
  - *T + T'*: *{ inl(t): t ε T} U {inr(t'): t' ε T'}* with canonically induced equivalence
  - *t*: Type bound to t in context

Basics: Types...

- continued:
  - $\mu t.T$: smallest *per* X such that $X = T[X/t]$
  - $Bag(T)$: $\{ [v_1 \ldots v_n]: v_i \in T\}$ where $[v_1 \ldots v_n] =_{Bag(T)} [w1 \ldots wn]$ if $v_i =_T w_{\pi(i)}$ for some permutation π for all $i=1..n$.
  - $Set(T)$: $\{[v_1 \ldots v_n]: vi \in T\}$ where $[v_1 \ldots v_n] =_{Set(T)} [w_1 \ldots w_m]$ if:
    - for all i there exists j such that $v_i =_T w_j$, and
    - for all j there exists i such that $v_i =_T w_j$.

Example equivalences:

- Consider the sequence *"Jann".* It is an element of *Seq(Char), Bag(Char)* and *Set(Char):*
  - As element of *Seq(Char)* it is equivalent to "*Jann*", but neither *"nJan"* nor *"Jna".*
  - As element of *Bag(Char)* it is equivalent to *"Jann"* and "nJan", but not "Jna".
  - As element of *Set(Char)* it is equivalent to *"Jann", "nJan"*, and *"Jna".*
- $[[4, 9, 4], [1, 4, 4], [9, 4, 4, 9], [4, 1]] =_{Set(Set(int)} [[1, 4, 1], [9, 4, 9, 9, 4]]$

Discriminator

- A discriminator for type T is a function $D[T]: \forall t.\ Seq(T * t) \rightarrow Seq(Seq(t))$ such that, if $D[T][(l_1,v_1),\ldots,(l_n,v_n)] = [V_1,\ldots,V_k]$:
  - $V_1 \ldots V_k$ is a permutation of $[v_1,\ldots, v_n]$;
  - Iff $l_i =_T l_j$ then there is a block V_h that contains both v_i and v_j.

Top-down Discrimination

- Polytypic definition of discriminators:
  - $D[T]\ [(l_1,v_1)] = [[v_1]]$ for any T (* *Note: O(1)!* *)
  - $D[A]\ xss = D_A\ xss$ (given discriminator for A)
  - $D[1]\ [(l_1,v_1),\ldots,(l_n,v_n)] = [[v_1,\ldots, v_n]]$
  - $D[T{*}T']\ [((l_{11}\ ,\ l_{12}),v_1),\ldots,\ ((l_{n1}\ ,\ l_{n2}),v_n)] =$

    let $[B_1,\ldots,B_k] = D[T]\ [(l_{11}\ ,\ (l_{12},v_1)),\ldots,\ (l_{n1}\ ,\ (l_{n2},v_n))]$

    let $(W_1,\ldots,W_k) = (D[T']\ B_1,\ \ldots,\ D[T']\ B_k)$

    in $concat\ (W_1,\ldots,W_k)$

Top-down discrimination...

- Polytypic definition contd.:
  - $D[T+T']\ xss =$
    let $(B_1, B_2) = splitTag\ xss$
    let $(W1, W2) = (D[T]\ B_1, D[T']\ B_2)$
    in $concat\ (W1, W2)$
  - $D[t]\ xss = D_t\ xss$ where D_t is discriminator bound to t in context
  - $D[\mu t.T]\ xss = D[T]\ xss$ in context where t is bound to $D[\mu t.T]$ (recursive definition!)

Discriminator combinators

- Note that the definitions of *D[T+T']* and *D[T*T']* require *D[T]* and *D[T']* only
- Thus for each type constructor **, +* we can define a corresponding *discriminator combinator,* also denoted by **, +* that compose given discriminators for *T*, and *T'* to discriminators for *T*T'* and *T+T',* respectively.
- **Note**: Combinators are ML-typable, except for recursively defined ones (require polymorphic recursion)

Example: Sequence discriminator

- $D[Seq(T)] = D[\mu t.\ 1 + T * t] =$
  $= D[1 + T * t] \text{ with } t := D[Seq(T)]$
  $= D[1] + D[T*t] =$
  $= D[1] + D[T] * D[Seq(T)]$
- That is, $D[Seq(T)] = f$ where f is recursively defined: $f = D[1] + D[T] * f$
- E.g., *D[Seq(Char)]* is the canonical string discriminator.

Discrimination for bags and sets

- We can discriminate for bag equivalence by:
  - sorting the input labels (each of which is a sequence) according to a common sorting order, then
  - eliminating successive equivalent elements (for set equivalence only), and
  - applying ordinary sequence discrimination to the thus sorted sequences

Weak sorting

- Weak sorting sorts each sequence in a multiset according to some common sorting order.
- Basic idea:
  - Associate each element with all the sequences it occurs in.
  - Then traverse the elements and add them to their sequences.
  - In this fashion all sequences will contain their elements in the same order.

Optimal discrimination

- **Theorem**: *D[T] xss* executes in time O(|xss|) for all type expressions *T.*
- **Observation**: The discriminators need not always inspect all the input since discrimination stops as soon as a singleton equivalence class is identified.

Applications:

- *D[Seq(Char)]:* Finding unique words and all their ocurrences in a text
- *D[Bag(Char)]:* Finding the anagram classes of a dictionary (set of words)
- *D[μt. 1 + Bag(t) + (t * t)]:* Discrimination of simple type expressions under associativity and commutativity of product type constructor in linear time (Zibin, Gil, Considine [2003], Jha, Palsberg, Shao, Henglein [2003])
- *D[μt. (String * Bag(t)) + (String * Set(t)) + (String *Seq(t))]:* Discriminating terms with associative, associative-commutative and associative-commutative-idempotent operators in linear time (word problem)

Bottom-up discrimination

- Top-down discrimination is optimal for *unshared* data.
- Consider a dag defined by:
  $n'_0 = (n_1, n_1), n_0 = (n_1, n_1)$
  $n_1 = (n_2, n_2)$
  ...
  $n_k = ((), ())$
- Treating this as an element of
  μt. (t+1) * (t+1) (trees!) would require time $O(2^k)$.

Bottom-up discrimination

- The problem is that shared data (nodes, boxes, references) may occur in multiple calls during top-down MSD.
- Basic idea:
  - Stratify nodes into ranks according to their heights in the dag.
  - Discriminate (partition) *all* nodes of the same rank in one go. Do this in a bottom up fashion since discrimination of rank *k* nodes requires discrimination according to rank *k-1* nodes.

Bottom-up discrimination

- Extend the type language with *Box(T)* (pointers to values of type *T* under value equivalence)
  and *Ref(T)* (pointers to values of type T with pointer equivalence)
- **Theorem**: *D[T] S xss* for store (graph) *S* and input sequence *xss* executes in time and space $O(|S| + |xss|)$.

Applications:

- *D[μt. Box(Seq(String * t)) * Bool)]:* Minimization of acyclic finite state automata (Revuz [1992], Cai/Paige [1995])
- Construction of Reduced Ordered Binary Decision Diagrams (ROBDD) without hashing (Henglein [2005])
- Compacting garbage collection (Ambus [2004], see plan-x.org)
- Type-directed pickling (Kennedy [2004], Elsman [2004])
- Compacting garbage collection (Appel/Goncalves [1993])

References (Acyclic MSD):

Paige, Tarjan, ``Three Partition Refinement Algorithms'', SIAM J. Computing, 16(6):973-989, 1987 (Section 2: lexicographic sorting)

Cai, Paige, ``Look Ma, no hashing, and no arrays neither'', POPL 1991 (applications of string msd)

Cai, Paige, ``Using multiset discrimination to solve language processing problems without hashing'', TCS 145(1-2):189-228, 1995 (based on POPL 1991 paper)

References...

Paige, ``Optimal translation of user input in dynamically typed languages'', unpublished manuscript, 1991 (weak sorting, bag/set equivalence, bottom-up msd for trees and dags)

Paige, ``Efficient translation of external input in a dynamically typed language'', Proc. 13th World Computer Congress, Vol. 1, 1994 (optimal-time preprocessing of serialized input into internal data structures)

References...

Paige, Yang, ``High level reading and data structure compilation'', POPL 1997 (underpinnings and refinement of efficient preprocessing)

Zibin, Gil, Considine, ``Efficient algorithms for isomorphisms of simple types'', POPL 2003 (application of basic msd to isomorphism with distributivity)

References (Cyclic MSD):

Note: Term ``MSD'' not used in works below.

Downey, Sethi, Tarjan, ``Variations on the common subexpression problem'', JACM 1980 (list equivalence in cyclic graph)

Cardon, Crochemore, ``Partitioning a graph in *O(|A| log |V|,* TCS 1982 (bag equivalence in cyclic graph)

Paige, Tarjan, ``Three Partition Refinement Algorithms'', SIAM J. Computing, 16(6):973-989, 1987 (Section 3: coarsest partition refinement; set equivalence in cyclic graph)

Conclusions

- Optimal discriminators that can be generated automatically from definition of equivalence relation (can be extended to richer language for equivalence classes)
- Note: No pointers required!
- Practical performance of handcoded MSD typically comparable with hashing (in some cases better)
- References in strongly typed languages can be made discriminable without making them comparable or hashable

Discussion

- MSD techniques (historically for strings and graphs) can be "disassembled" into atomic components (*, +, μ,…) and then orthogonally combined freely to arrive isassembly of MSD-techniques
- Identification of type of discriminators has been crucial for admitting inductive/polytypic definition of discriminators
- Discriminators stress ML-polymorphism: Reference discrimination (semantically safe side effects, but prohibited by ML reference typing) and discrimination for recursively defined types (polymorphic recursion required)
- Reference discrimination (instead of equality) would be an easy useful extension to ML without performance or semantic penalties, yet support for linear-time discrimination (presently requires $O(n^2)$ time using reference equality alone).
- Discriminators can be extended to cyclic data at cost of *log(n)* factor. Requires more refined algorithmic techniques.

Open questions

- Automatic generation of *efficient* (not handcoded) discriminators ; e.g., by partial evaluation
- Algorithm engineering: I/O, cache-sensitivity analysis
- Empirical evaluation of MSD in a variety of applications (e.g., ROBDDs, coalescing garbage collection, run-time verification, type checking
- Identification of scenarios where 'weak' machine model required by MSD is an advantage
- Extension of MSD to scoped values (e.g., alpha-congruence), other extensions

More information

- Paper under preparation.
  See www.plan-x.org/msd