

# Multiset Discrimination

Fritz Henglein\*

September 16th, 2003

## Abstract

Multiset discrimination is a fundamental technique for finding duplicates in linear time without hashing or comparison-based sorting. It can be viewed as a generalization of equality (or equivalence) testing from two arguments to an arbitrary number of arguments since it decides all the pairwise equalities between its inputs in one go by grouping them into equivalence classes.

In this paper we provide a general framework for multiset discrimination suitable for packaging multiset discriminators as a reusable software component. It shows how multiset discriminators can be defined *polytypically*; that is, inductively on the type structure of the input data. The polytypic discriminators are optimal for data structures without sharing. We show how linear time multiset discriminators can be defined for shared, acyclic data. Finally, we point out that three seemingly different algorithms on partition refinement for circular solve certain instances of multiset discrimination for We conclude by pulling them together into a single algorithm

This allows extending multiset discrimination to abstract data types and type constructors and suggests that multiset discrimination should be built as base functionality into types, generalizing equality. The algorithmic ingredients behind multiset discrimination have been published before, though under disparate names and for special instances of multiset discrimination. Our contribution lies in demonstrating that can be combined for multiset discrimination in basically arbitrary cyclic data structures in time  $O(m \log n)$  for data structures with  $m$  edges and  $n$  nodes.

We provide general considerations for applying multiset discrimination vis a vis hashing and (comparison-based) sorting and give some empirical evidence of its practical efficiency.

---

\*Affiliation: DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, Email: henglein@diku.dk. This research was partially supported by the Danish Research Council under Project *PLI*.

# 1 Introduction

Imagine you are asked to (write a program to) determine the anagram classes in a dictionary; or decide whether two sequences of integers represent the same set or multiset; or minimize an acyclic deterministic finite state automaton; or replace all distinct tokens in a program by numbers  $0, 1, \dots, n$  (but same tokens by same numbers, of course); or figure out whether two nonrecursive types are isomorphic up to associativity and commutativity of the product type constructor; or decide the word problem for terms with binary operators that may be associative, commutative, idempotent or any combination of that. Imagine also that the program should be efficient in practice and run in worst-case linear time, without hashing or any dynamic allocation of arrays for that matter. And, while you are at it, your program might as well compact the input by eliminating all duplicates or isomorphic elements for space savings and for enabling improved performance of subsequent operations on those data.

Or imagine you are asked to compress graphs or multigraphs (with ordered or unordered sets of neighbors with  $n$  nodes and  $m$  edges by identifying all isomorphic nodes in it and to do so in time  $O(m * \log n + n)$ , again without hashing. In other words do the above problems, but allowing cyclic data structures.

All these problems, and many more, can be solved by multiset discrimination, a set of algorithmic techniques developed by Paige *et al.* in 1984-1997. Multiset discrimination is a batch-oriented technique for solving equivalence problems. Since it can supplant both hashing and (comparison-based) sorting in many contexts, has better asymptotic complexity than both of them, and is easily implemented in a modular fashion without breaking abstraction barriers, we argue that it should be considered a basic algorithmic tool on a par with hashing and sorting.

## 2 Basics

In this section we introduce the basic concepts.

### 2.1 Atomic types

Let  $A$  be a set of *atomic type names* with an associated *type interpretation*, which maps each  $a \in A$  to a triple  $(V_a, |\cdot|_a, =_a)$  where:

1.  $V_a$  is a set of *atomic values*, the values of type  $a$ ;
2.  $|\cdot|_a$  maps each element of  $V_a$  to a natural number, its *size*; and
3.  $=_a$  is an equivalence relation on  $V_a$ , which we call the *equality* on  $V_a$  (or simply  $a$ ).

We assume that  $V_a, V_b$  are disjoint for  $a \neq b$ . We shall refer to  $A$  and its type interpretation simply by  $A$ .

## 2.2 Values and their sizes

The set  $\mathcal{V}^A[N]$  of *values* over atomic types  $A$  and node set  $N$  is defined to be the set of expressions (abstract syntax trees) given inductively by the following grammar.

$$v ::= () \mid (v_1, v_2) \mid \text{Inl}(v_1) \mid \text{Inr}(v_2) \mid c \mid n$$

where  $c$  ranges over  $\bigcup_{a \in A} V_a$  and  $n$  ranges over elements from the set of *nodes* (or *locations* or *pointers*)  $N$ , which we assume to be disjoint from atomic values.

The *size*  $|v|$  of a value  $v$  is defined inductively as follows:

$$\begin{aligned} |()| &= 0 \\ |(v_1, v_2)| &= |v_1| + |v_2| \\ |\text{Inl}(v_1)| &= 1 + |v_1| \\ |\text{Inr}(v_2)| &= 1 + |v_2| \\ |c| &= |c|_a \text{ for } c \in V_a \\ |n| &= 1 \end{aligned}$$

The size function is intended to give a measure of how much memory a value requires when stored. Note the following:

- The size function reflects a unit-cost pointer model, where nodes (store addresses) are considered to have unit size and allow constant-time access. For algorithmic analysis, this simplifying assumption is justified as long as the number of store addresses we require in an algorithm is reasonably related to the input size of the problem given problem instance; see e.g. Tarjan [Tar83].
- The size of a pair is the sum of the sizes of each component, even if the components are identical. The size function measures the space required to store a value in *unboxed* (*endogenous*) or in *boxed* (*exogenous*) form with *linear* pointers only; that is, without sharing. (If a value occurs more than once in another value, it needs to be stored multiple times to ensure that there is but one pointer to each occurrence of it.) Only *nodes* are *sharable* pointers.

### 2.3 Type expressions

The set  $\mathcal{T}^A$  of (*first-order*) *type expressions* over atomic types  $A$ , henceforth simply called *types*, is defined to be the set of expressions (abstract syntax trees) given inductively by the following grammar.

$$\tau ::= 1 \mid \tau \times \tau \mid \tau + \tau \mid a \mid \text{ref}(\tau) \mid \alpha \mid \mu\alpha.\tau$$

where  $a \in A$ ,  $\alpha$  ranges over a denumerable set of *type variables* disjoint from  $A$ ,  $\mu\alpha.\tau$  binds  $\alpha$  in  $\tau$ , and  $\tau$  must not be a type variable. A type  $\tau$  is *closed* if it has no free type variables.

We define the  $k$ -fold product  $\tau^k$  as follows:

$$\tau^k = \begin{cases} 1 & \text{if } k = 0 \\ \tau \times \tau^{k-1} & \text{if } k > 0 \end{cases}$$

We can define the list type constructor as type scheme; that is a type with a free type variable:  $\text{list}(\alpha) = \mu\beta.1 + \alpha \times \beta$ . Other type constructors can be define similarly.

We abbreviate  $\text{Inr}(v_1, \text{Inr}(v_2, \dots, \text{Inr}(v_k, \text{Inl}()) \dots))$  as  $[v_1, v_2, \dots, v_k]$ . It is easy to see that all elements of type  $\text{list}(\tau)$  for any  $\tau$  must have this form.

### 2.4 Stores

A *store*  $S$  is a pair  $(N, E)$  where  $N$  is a finite set of nodes and  $E$  is a mapping from  $N$  to  $\mathcal{V}^A[N]$ . We call  $\{E(n) : n \in N\}$  the *values stored* in  $S$ . Note that each node that occurs in a stored value must in turn be mapped to some value (“no dangling pointer”), any given node may occur multiple times (sharing), and stores may form cyclic data structures. occur multiple times (sharing). We can think of

The size  $|S|$  of a store  $S = (N, E)$  is defined as

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

### 2.5 Store typings

A *candidate store typing*  $\Gamma$  for store  $S = (N, E)$  is a set of pairs of the form  $(n, \text{ref}(\tau))$ , where  $n \in N$  and  $\tau$  is a type. We say value  $v$  has type  $\tau$  under candidate store typing  $\Gamma$ , written  $\Gamma \models v : \tau$ , if  $\Gamma \vdash v : \tau$  can be derived in the inference system of Figure 1.

We can now define the *store typing*  $\Gamma^S$  of  $S = (N, E)$  coinductively as the largest candidate store typing  $\Gamma$  for  $S$  for which we have  $\Gamma \models E(n) : \tau$  for all  $(n, \text{ref}(\tau)) \in \Gamma$ . Note that  $\Gamma^S$  is well-defined since  $\Gamma \models v : \tau$  is monotonic in  $\Gamma$ ; that is, if  $\Gamma \subseteq \Gamma'$  and  $\Gamma \models v : \tau$  then  $\Gamma' \models v : \tau$ . Finally, we

$$\Gamma \vdash () : 1$$

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

$$\frac{\Gamma \vdash v_1 : \tau_1}{\Gamma \vdash \text{Inl}(v_1) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \text{Inr}(v_2) : \tau_1 + \tau_2}$$

$$\Gamma \vdash c : \tau_c \text{ (if } \tau_c \text{ is atomic type of } c\text{)}$$

$$\Gamma \vdash n : \text{ref}(\tau) \text{ (if } (n, \text{ref}(\tau)) \in \Gamma\text{)}$$

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

Figure 1: Value typing rules

say  $v$  has type  $\tau$  under store  $S$  and write  $S \models v : \tau$  if  $\Gamma^S \models v : \tau$ . We say  $v$  has type  $\tau$  or is an *element* of  $\tau$  under  $S$ . If  $S$  is irrelevant (for values with no nodes) or understood from the context, we will just say that  $v$  has type  $\tau$ .

## 2.6 Equivalence relations on nodes

An *equivalence relation* on set  $X$  is a binary relation  $\mathcal{E} \subseteq \mathcal{X} \times \mathcal{X}$  that is reflexive, symmetric and transitive. It *partitions*  $X$  into a set of disjoint *equivalence classes* (or *blocks*) where  $x, y$  belong to the same block if and only  $(x, y) \in \mathcal{E}$ . Equivalence relations on  $X$  constitute a distributive lattice under operations  $\vee$  and  $\wedge$  where  $\mathcal{E}_1 \vee \mathcal{E}_2$  is the least equivalence relation containing  $\mathcal{E}_1 \cup \mathcal{E}_2$  and  $\mathcal{E}_1 \wedge \mathcal{E}_2 = \mathcal{E}_1 \cap \mathcal{E}_2$ . For distinct equivalence relations  $R, R'$  on  $N$  we say  $R$  is a *refinement* of  $R'$  or  $R'$  is *coarser* than  $R$  if  $R \leq R'$  in this lattice. We write  $\perp_N = \{(n, n) : n \in N\}$  (or simply  $\perp$  if the  $N$  is clear from the context) for its bottom element, and  $\top_N = \{(n, n') : n, n' \in N\}$  (or simply  $\top$ ) for its top element.

Given store  $S = (N, E)$ , let  $\mathcal{E}$  be an equivalence relation on  $N$ . We write  $\cong_{\mathcal{E}}$  for the *congruence* induced by  $\mathcal{E}$  on  $\mathcal{V}^A[N]$ ; that is, the smallest binary relation  $R$  with following closure property (*compatibility*):

1.  $(c, c') \in R$  for all  $c, c' \in V_a$  such that  $c =_a c'$ ;
2.  $((), ()) \in R$ ;
3. if  $(v, v') \in R$  and  $(w, w') \in R$  then
  - (a)  $((v, w), (v', w')) \in R$ ,
  - (b)  $(Inl(v), Inl(v')) \in R$ , and
  - (c)  $(Inr(w), Inr(w')) \in R$ .

Note that  $\cong_{\mathcal{E}}$  is an equivalence relation. We write  $v = v'$  if  $v \cong_{\perp} v'$ .

An equivalence relation  $\mathcal{E}$  on  $N$  induces a (not necessarily different) equivalence relation by relating the values of the nodes: Define  $Equiv_S(\mathcal{E}) = \{(n, n') : n, n' \in N \wedge E(n) \cong_{\mathcal{E}} E(n')\}$  for store  $S = (N, E)$ . We say  $\mathcal{E}$  induces  $Equiv_S(\mathcal{E})$  and call  $Equiv_S(\mathcal{E})$  the *equivalence induced by  $\mathcal{E}$* . With this in place, we can now define the *isomorphism* relation  $\cong_S$  on a store  $S = (N, E)$  to be the largest equivalence relation on  $N$  which induces itself; that is, it is the largest relation  $R$  such that  $R = Equiv_S(R)$ . Note that  $\cong_S$  is well-defined since  $Equiv_S(\cdot)$  is monotonic with respect to  $\leq$  in its argument. In particular,  $\cong_S$  can be computed by greatest fixed point iteration since it is  $\equiv_S^i(\top)$  for some  $i \in \mathcal{N}$ . Note that  $\cong_{\mathcal{E}}$  is independent of  $\mathcal{E}$  on  $V^A = V^A[\emptyset]$ . Consequently we may write, ambiguously but correct,  $v \cong v'$  if  $v \cong_{\mathcal{E}} v'$  for such values.

## 2.7 Implementation model

In the following we shall be informal (as is usual) when performing algorithmic analysis of our.

Values are, as mentioned above, implemented as (proper) tree structures with atomic values (incl. ()) and nodes at the leaves. Each internal tree node has but one pointer to it. If a value is to be stored more than once this requires copying its tree data structure.

A store  $S$  is, as the name suggests, implemented as a set of memory locations (the nodes). Such a memory location contains a (singly-referenced) pointer to the tree representation of its value.

We can think of the nodes in a data structure as classified into *linear* nodes and *nonlinear* nodes: the linear nodes are pointed to exactly once, and the nonlinear ones may be pointed to any number of times. Note that the nonlinear ones (only) correspond to our node set  $N$  in a store  $S = (N, E)$ . The linear ones are implicit in the values that  $E$  maps the nodes to. In this fashion all sharing and cyclicity of data structures can only occur through  $N$ .

## 3 Multiset discrimination: The problem

Informally, the term (*multiset discrimination* refers to the problem of partitioning a list of input values into equivalence classes of pairwise equal or equivalent elements. *Multiset* refers to the input list and our intention of viewing it as a representation of a multiset, where the order of elements is irrelevant. *Discrimination* refers to the process of distinguishing (discriminating) between nonequal (or nonequivalent) elements, the result of which is expressed by partitioning the input into equivalence classes.

Note that multiset discrimination applied to a list of two elements decides equality (or equivalence) between the two elements: if the result has but one equivalence classe, the two elements are equal (equivalent); if it has two equivalence classes they are not. In this sense multiset discrimination can be viewed as the generalization of equality (equivalence) testing from two arguments to any finite number of arguments.

We shall generalize the definition of multiset discrimination somewhat. We shall split each value in the input list into two parts: a *label* and an *information item*. A discriminator then collects those information items with equivalent labels into equivalence classes and returns them. In doing so, it inspects only the labels (which may be arbitrarily structured values) without accessing the information items themselves; that is, it is *parametric polymorphic* in the information items. The labels themselves are not part of the output.

The fact that the labels in the input are not returned in the output is a convenience. It makes for simple definition and an efficient implementation

of discriminators since discriminators can freely destruct and discard labels during processing without having to store them. This does not constitute a loss of expressiveness: If we want to retain the labels as part of the output, we simply call the discriminator on input  $[(v_1, (v_1, w_1)), \dots, (v_n, (v_n, w_n))]$  instead of  $[(v_1, w_1), \dots, (v_n, w_n)]$ .

We shall make the above precise now. We say  $f$  has type  $\tau \rightarrow \tau'$  if  $f$  terminates with a value of type  $\tau'$  for each input value of type  $\tau$ . Apart from its input,  $f$  may access and update a store equipped with one or more equivalence relations;  $f$  has type  $\forall\beta.\tau$  if it is parametric polymorphic in  $\beta$  [Rey83, Wad89]. For a list of lists  $P = [B_1, \dots, B_k]$  we write  $\text{concat}(P)$  for the concatenation of  $B_1, \dots, B_k$ . If  $R$  is a list of pairs, we write  $\text{Domain}(R)$  for the list of first components of the elements of  $R$  and  $\text{Range}(R)$  for the list of their second components, in both cases in the same order as in the input. Finally, we write  $R_x^\sim(R)$  for the list of second components  $w$  of elements  $(v, w)$  where  $v \sim x$ , again in the same order as they appear in  $R$ .

**Definition 3.1** [Discriminator] A *discriminator* for  $\tau$  under equivalence relation  $\sim$  is a parametric polymorphic function  $D_\tau : \forall\beta.\text{list}(\tau \times \beta) \rightarrow \text{list}(\text{list}(\beta))$  that operates on a store (implicit or global argument) such that

1.  $\text{concat}(D_\tau(R))$  is a permutation of  $\text{Range}(R)$ ; and
2. If  $D_\tau(R) = [B_1, \dots, B_k]$  then for each  $i, 1 \leq i \leq k$  there is  $x \in \text{Domain}(R)$  such that  $B_i$  is a permutation of  $R_x^\sim$ .

If the latter property holds with the identity permutation, that is  $P_i = R_x^\sim$ , we say the discriminator is *order-preserving*.  $\square$

Note that labels need not be atomic. They may be arbitrarily complex values. Without loss of generality, we assume that the second components of the input are passed as nodes.

## 4 Atomic discrimination

The above defines the *problem* of multiset discrimination. The term *multiset discrimination*, however, has been introduced by Paige *et al.* to refer to a set of particular algorithmic techniques for multiset discrimination that avoid hashing (and, coincidentally, comparison-based sorting).

We capture these techniques in stages. First we cover *basic multiset discrimination* for nodes and atomic types. Then we extend them by polytypic definition to types constructed from products, sums and uniform recursion.

### 4.1 Numbers

Consider the atomic type  $\overline{K}$ , which denotes the initial segment of numerals  $[0 \dots K - 1]$  with  $|v|_{\overline{K}} = 1$  and  $v_1 =_{\overline{K}} v_2$  if and only if  $v_1$  and  $v_2$  denote the same numbers.

Given numerals  $V = [v_1, \dots, v_l]$  drawn from  $\overline{K}$  We can implement multi-set discrimination  $\Delta_{\overline{K}}$  for  $\overline{K}$  under equality  $=_{\overline{K}}$  by using a *single*, statically allocated and initialized *dictionary object*  $DictObj_K$  with the following fields and methods:

*dictArray*: array of size  $K$ , indexed by  $[0 \dots K - 1]$ ;

*labels*: field for holding a list with elements in  $[0 \dots K - 1]$ .

*add(x, y)*: method which adds  $y$  at the end of the list at index  $x$  in *dictArray*; furthermore, if the list at  $x$  is empty before that, it adds  $x$  to *labels* (anywhere).

*getLists()*: returns the list of all nonempty lists in *dictArray* by looping through the indexes stored in *labels*. In in this process, all *dictArray[x]* are reset to the empty list, and, finally, *labels* is also reset to the empty list.

Using *DictObj* as a global variable bound to  $DictObj_K$ , we can implement  $\Delta_{\overline{K}}[(v_1, w_1), \dots, (v_l, w_l)]$  as follows:

1. For each  $i \in [1 \dots l]$ , add  $(x_i, y_i)$  to *DictObj* (that is, call *DictObj.add(x<sub>i</sub>, y<sub>i</sub>)*).
2. Return the nonempty lists in  $DictObj_K$  and reset it; that is, call *DictObj.getLists()*.

**Proposition 4.1**  $\Delta_{\overline{K}}$  is an order-preserving discriminator for  $\overline{K}$  under  $=_{\overline{K}}$  that executes in time  $O(n)$  on input  $[(v_1, w_1), \dots, (v_n, w_n)]$ .

Note that this holds even if  $n \ll K$  since we have assumed  $DictObj_K$  to be allocated and initialized at program start time (at zero cost if  $K$  is static, or at cost  $O(K)$  if  $K$  is an input parameter). In other words, array and The key invariant is that before and after every call to  $\Delta_{\overline{K}}$  both *labels* and *dictArray[x]* for each  $x \in [0 \dots K - 1]$  are empty.

Allocation of  $DictObj_K$  requires time  $O(K)$  since each element of the array needs to be initialized to the empty list. By a well-known coding trick [AHU74, Exercise 2.12], initialization can be reduced to  $O(1)$  time. It is based on  $O(1)$ -time reservation of memory for arbitrary sized arrays, which are then assumed to be unpredictably initialized. Treating the allocation as an  $O(1)$  operation is problematic from both theoretical and practical viewpoints in some situations. It is justifiable as long as the size of the allocated array is reasonably related to the input size of the problem instance that is being solved. If we use a number that is input to a procedure to allocate an array of that size, the array is exponentially bigger than the input size (in logarithmic-cost model), respectively arbitrarily larger than the input size (in a unit-cost model). Here, treating the array allocation as a constant-time operation is, from a physical viewpoint, questionable.

In practical terms the problem is quickly evident. Imagine we want to discriminate a list of  $n$  64-bit integers on a contemporary computer. This can be done in time  $O(n)$  using  $\Delta_{\frac{2^{64}}{2}}$ ; that is, using a static array of size  $2^{64}$ , which considerably exceeds the cumulative digital electronic storage there has ever been in existence. We may choose to improve upon this by only allocating an array of size  $M$ , where  $M$  is the maximum value in the input, but this doesn't help much since  $11\dots111$  may be in the input. A final note: Implementing the coding trick in modern type safe programming languages is impossible since, as experience has shown, allowing uninitialized or unpredictably initialized storage structures is a bad idea.

Interestingly, Paige and Tarjan make use of this trick in the multiset discrimination phase of their lexicographic sorting algorithm since they believe that a new  $DictObj_K$ -objects need to be allocated dynamically at the beginning of a partition step [PT87, p. 976]. This, however, is unnecessary: a single  $DictObj_K$  object is sufficient. This simplifies their algorithm and makes their results immediately applicable on a machine model that requires  $\Theta(n)$  time for allocation of arrays of size  $n$ .

Zibin, Gil and Considine use the trick to discriminate  $n$  integers from  $\overline{K}$  in time  $O(n)$  and  $\Theta(K+n)$  space [ZGC03, Lemma 2.5]. As we have pointed out, this is asymptotically questionable for  $n \ll K$ . We shall see how  $\overline{K}$  and even  $\mathcal{N}$ , the infinite set of all nonnegative integers, can be discriminated space and time efficiently, using multiset discrimination for product types.

We shall assume henceforth that we only have one atomic type of the form  $\overline{K}$  in  $A$  with a "small"  $K$ : the type  $char = \overline{256}$  of *characters*. Indeed, this is only for practical convenience since even  $char$  can be coded and discriminated as the type  $2^8$  where  $2 = 1 + 1$ . Since, in the following,  $\Delta_{\overline{K}}$  is the only discriminator that applies arrays and address arithmetic this means that multiset discrimination ultimately only requires pointer operations.

## 4.2 References

Given store  $S = (N, E)$  with equivalence relation  $\mathcal{E}$  on  $N$  and an input list  $R = [(v_1, w_1), \dots, (v_n, w_n)]$  where the  $v_i$  are references, we would like to discriminate  $R$  under  $\mathcal{E}$ . In the canonical case where  $\mathcal{E} \perp_N$ , which relates any reference only to itself, this corresponds to grouping the input into information items  $w_i$  with equal references as labels.

We can represent an equivalence relation on a store by implementing references as pointers to records containing both (the tree representation of) its value and an auxiliary pointer that represents the equivalence class in  $\mathcal{E}$  that the reference belongs to. We call this auxiliary pointer the *equivalence class pointer* or  $\mathcal{E}$ -*pointer* of the reference. (In general, a record have more than one auxiliary pointer, one for each equivalence relation we are interested in maintaining on references. For now, we restrict ourselves to records with but one equivalence class pointer.) The  $\mathcal{E}$ -pointer points to

some auxiliary storage area we only need during multiset discrimination of references. It plays a dual role:

- It allows deciding equivalence in constant time: Two nodes are equivalent if their  $\mathcal{E}$ -pointers are equal (as pointers).
- It enables basic multiset discrimination on nodes: It points to a location in auxiliary memory for storing lists of values during multiset discrimination. It is empty before and after that.

A *discrimination record* *discObj* is a record with the following fields:

*contents*: field holding the *contents* of the record;

*ecPointer*: field holding an equivalence class pointer.

Each equivalence class pointer points to a *discrimination object* (or  $\mathcal{E}$ -object) which has the following field and methods:

*assocInfo*: (private) field for storing arbitrary information, in particular for storing lists of discrimination record pointers;

*isEmpty()*: returns true if and only if *assocInfo* is empty;

*add(y)*: method that adds *y* to the end of *assocInfo*;

*getList()*: method that returns the contents of *assocInfo* before resetting it to the empty list.

A *discriminable reference* is a reference to a discrimination record.

Discriminable references and their contents encode the global store  $S = (N, E)$  and  $\mathcal{E}$ : There is a discriminable reference for each node in  $N$ ;  $E(n)$  is stored as the contents of the discriminable object that  $n$  references; and  $(n, n') \in \mathcal{E}$  holds if and only if the contents of the *ecPointer*-fields of  $n$  and  $n'$  are equal. Since both *contents* and *ecPointer* are accessible and updateable, this means updates to  $E$  and  $\mathcal{E}$  are effected by updating existing objects, and updates to  $N$  can be performed by allocating new discriminable objects or deleting existing ones.

We can implement  $\Delta_{ref(\cdot)}[(v_1, w_1), \dots, (v_n, w_n)]$  as follows, if  $v_1, \dots, v_n$  are implemented by discriminable references:

1. Assume *ecPointers* = [];
2. For each  $i \in [1 \dots n]$  do:
  - (a) if  $v_i.ecPointer.isEmpty()$  then *ecPointers* :=  $v_i.ecPointer :: ecPointers$ ;
  - (b)  $v_i.ecPointer.add(w_i)$ ;
3. *res* := [];

4. For each  $p \in ecPointers$  do:  
 $res := p.getList() :: res;$
5.  $ecPointers := [];$
6. Return  $res;$

**Proposition 4.2**  $\Delta_{ref(\cdot)}$  is an order-preserving discriminator for  $ref(\tau)$  for any type  $\tau$  under  $=_{\mathcal{E}}$  that executes in time  $O(n)$  on input  $[(v_1, w_1), \dots, (v_n, w_n)]$ .

It is easy to underestimate the broad applicability of discrimination pointers in general. They not only allow constant-time testing of equivalence, they make associating of arbitrary information with an equivalence class in worst-case constant time possible. We have used this to associate all the information items with labels in the same equivalence class.

## 5 Structured value discrimination

In the previous section we have seen efficient order-preserving discriminators for atomic type *char* (more generally for initial segments  $[0 \dots K - 1]$ ) and for references. In this section we show how to extend them to discriminators for arbitrary first-order types build from product, sum and inductive types defined by uniform (type) recursion.

More precisely, let us write  $\delta\tau$  for the *discriminator type*  $\forall\beta.list(\tau \times \beta) \rightarrow list(list(\beta))$ . Figure 2 defines a family of functions  $\Delta_{\tau}^{\Gamma}$  of type  $\delta\tau$  for each  $\tau$ , where  $\Gamma$  maps each type variable  $\alpha$  occurring free in  $\tau$  to a function  $\Gamma(\alpha)$  of type  $\delta\alpha$ .

Note that  $\Delta_{\tau}^{\Gamma}$  is defined *polytypically*, that is by induction on the type structure of  $\tau$  [JJ96]. In particular, each  $k$ -ary type constructor is mapped to a  $k$ -ary discriminator constructor, which input discriminators for the input types of the type constructor and produces a discriminator for its result type. (In other words,  $\Delta$  is defined functorially.)

This lays out a blueprint for extending  $\Delta$  to other basic types and type constructors, including abstract types and type constructors. In particular, it suggests that providing a discriminator (discriminator constructor) for a type (type constructor) is as fundamental an operation on types as providing an equality operation and that this can be done without breaking the abstraction barrier of abstract types beyond making (abstract) equality observable. What makes this compelling is that  $\Delta$  is asymptotically optimal (linear time) and practically (easy to implement, low overhead) efficient.

Discriminator  $D_{\tau}$  is *optimal* if it executes in time  $O(n)$  and auxiliary space  $O(n)$  on inputs  $R$  with  $|R| = n$ .

**Lemma 5.1 (Optimality)** *If  $\Gamma$  maps each  $\alpha$  to an optimal order-preserving discriminator  $\Delta_{\alpha}$  that is optimal for type  $a$ , where  $a$  has type interpretation*

Let  $reassoc(\cdot)$ ,  $split(\cdot)$  be functions such that:

$$reassoc([(u_1, v_1), w_1], \dots, [(u_n, v_n), w_n]) = [(u_1, (v_1, w_1)), \dots, (u_n, (v_n, w_n))],$$

$$split(R) = [(v', w) \mid (v, w) \in R \wedge v = Inl(v')], [(v'', w) \mid (v, w) \in R \wedge v = Inr(v'')].$$

$$\begin{aligned} \Delta_1^\Gamma(R) &= [Range(R)] \\ \Delta_{\tau_1 \times \tau_2}^\Gamma(R) &= \text{let } R' = reassoc(R) \\ &\quad [S_1, \dots, S_k] = \Delta_{\tau_1}^\Gamma(R') \\ &\quad [W_1, \dots, W_k] = [\Delta_{\tau_2}^\Gamma(S_1), \dots, \Delta_{\tau_2}^\Gamma(S_k)] \\ &\text{in } concat([W_1, \dots, W_k]) \\ \Delta_{\tau_1 + \tau_2}^\Gamma(R) &= \text{let } (R_1, R_2) = split(R) \\ &\quad (W_1, W_2) = (\Delta_{\tau_1}^\Gamma(R_1), \Delta_{\tau_2}^\Gamma(R_2)) \\ &\text{in } concat([W_1, W_2]) \\ \Delta_\alpha^\Gamma(R) &= \Gamma(\alpha)(R) \\ \Delta_{\mu\alpha.\tau}^\Gamma &= f \text{ where} \\ &\quad f(R) = \Delta_\tau^{\Gamma\{\alpha \mapsto f\}}(R) \\ \Delta_a^\Gamma(R) &= \Delta_a(R) \text{ (where } \Delta_a \text{ is the standard discriminator for } a) \\ \Delta_{ref(\tau)}^\Gamma(R) &= \Delta_{ref(\cdot)}(R) \text{ (where } \Delta_{ref(\cdot)} \text{ is the standard discriminator for discriminable references)} \end{aligned}$$

Figure 2: Standard discriminators for structured types

$(V_a, |\cdot|_a, =_a)$ , then  $\Delta_\tau^\Gamma$  is an optimal order-preserving discriminator for type  $\tau$ .

**Theorem 5.2** *Let  $\tau$  be a closed type. Then  $\Delta_\tau^\Gamma$  is independent of  $\Gamma$ , which we simply write as  $\Delta_\tau$ . Furthermore,  $\Delta_\tau$ , when applied with global store  $S = (N, E)$  and equivalence relation  $\mathcal{E}$  on  $N$ , is an optimal order-preserving discriminator for  $\tau$  under  $\cong_{\mathcal{E}}$ .*

We refer to the type-indexed family of functions  $\Delta_\tau$  collectively as  $\Delta$ . We may also simply write  $\Delta$  as a short-hand for  $\Delta_\tau$  for the appropriate  $\tau$ . Note that the type-indexed nature of  $\Delta$  does not restrict  $\Delta$  to be applied to homogeneous sets of values only. All values in  $V^A[N]$  can be represented as elements of the universal type  $U = \mu\alpha.1 + (\alpha \times \alpha) + (\alpha + \alpha) + \Sigma_a a + \text{ref}(\alpha)$ , which is a definable type.

Recall that the type of lists over elements of type  $\beta$  can be defined as a type schema (type with at least one free variable):

$$\text{list}(\alpha) = \mu\alpha.1 + \beta \times \alpha.$$

In particular, we can define the type of strings *string* as  $\text{string} = \text{list}(\text{char})$ . Figure 2 then provides a standard discriminator for *string*. It discriminates a list of string inputs as labels and associated references to information items in worst-case linear in the sum of the lengths of the string labels.

The standard discriminator for lists processes the input informally as follows: It splits the input into occurrences of the empty list and occurrences of nonempty lists. The nonempty lists are then split into subclasses of lists according to their first elements by applying the element discriminator to them, splitting the nonempty lists into classes of lists. These classes are then discriminated by applying the discriminator recursively to each class separately. The discriminator stops the recursion as soon as a class contains only a single list element. In this fashion, the discriminator only looks at the *minimal distinguishing prefixes* of the input lists.

Since strings are isomorphic to lists of characters, this provides an efficient string discriminator where only the minimal distinguishing prefixes are processed. This is the essence of Paige/Tarjan's lexicographic sorting algorithm [PT87, Section 2], which refuted the then widely held belief that linear-time lexicographic sorting required right-to-left processing of the input strings and thus processing of *all* the characters in the input [Meh84, exercise 18, p. 100].

We can improve the implementation of  $\Delta$  by adding the clause

$$\Delta_\tau^\Gamma([(v, w)]) = [[w]]$$

to its implementation at each type  $\tau$ . This clause expresses that there is no need for processing the label  $v$  if the input consists of a single pair and

there can thus be only a single singleton equivalence class as the answer. This simple observation is at the center of the discrimination phase of Paige and Tarjan’s lexicographic sorting algorithm [PT87, Section 2]. For strings — or generally lists — it means that the discriminator only processes the minimum distinguishing prefixes of the string labels.

Multiset discrimination was first developed for strings [PT87, Section 2]. Our polytypic definition of  $\Delta$  factors its algorithmic ingredients into its independent basic components for the unit type, sums, products and recursive type definitions and thus generates discriminators for arbitrary first-order types, not just strings. We shall look at a number of special cases and applications below.

### 5.0.1 Large integers revisited

Fixed size integers, such 32-bit or 64-bit integers, can be viewed as strings. In particular, we can decompose a 32-bit integer uniquely in constant time into an element of  $char^4$ .  $\Delta_{char^4}$  then yields an optimal discriminator on such fixed size integers that requires only a single 256-element array — the one array More radically, a 32-bit integer can be treated “bit sequentially” as an element of  $2^{32}$ , and  $\Delta_{2^{32}}$  yields a discriminator that requires no array operations at all. (On contemporary general purpose computers decomposition into 8-bit blocks (characters) or 16-bit blocks (elements of  $65536$ ) is substantially more efficient, however.)

Infinite precision integers can be viewed as elements of *string* and discriminated in linear time by  $\Delta_{string}$ .

**Example 5.3** [Example discriminators] An efficient *char*-discriminator  $d_{char}$  can be used to produce a mapping from characters to all the strings that contain a given character in time linear in the size of the input (the sum of the sizes of the input strings). To do so, given a list of strings, form the list of all pairs  $(c, s_r)$  where  $c$  is a character occurring in string  $s$  and  $s_r$  is a pointer to  $s$ . Then apply  $d_{char}$  to that list. In the result, each character is associated with the list of (pointers to) strings containing that character.

An efficient *string*-discriminator  $d_{string}$  can be used to find all the occurrences of a word in a text and to so for all words in the text in time linear in the input text. To do so, simply execute  $d_{string}(l)$  where  $l'$  is a list of pairs  $(s, p)$  of strings  $s$  occurring at position  $p$  in the input text in document order. The result will list all the unique strings in the input (in the order they were first encountered), associated with a list of all the positions at which they occur. We could use hashing or sorting techniques to solve this problem. They will not qualify as *efficient* discriminators, however, since they do not run in worst-case linear time. We shall see an efficient string discriminator shortly.

□

## 6 Multiset discrimination under nontrivial equivalences

There are cases where we would like to discriminate up to some nontrivial equivalence relation on the labels. For example, the equivalence relating any two strings whose first 4 characters are equal, or the equivalence that relates strings  $s$  and  $s'$  if there is a permutation of the letters in  $s$  resulting in  $s'$ . In the first case, the equivalence is induced by a function, namely the function that maps a string to its prefix of up to the first 4 letters. In the second case, the equivalence is induced by an algebraic property: the associativity and commutativity of the string concatenation operator or, equivalently, by treating the string not as a sequence but as a *bag* (*multiset*).

We shall see treat two cases cases: equivalence relations that can be defined by functions, and equivalence relations induced by set- and bag-values. (Sequences are already covered as lists).

### 6.1 Function-induced equivalences

Let  $f$  be a function  $\tau \rightarrow \tau'$ . We write  $v_1 \cong_{\mathcal{E}}^f v_2$  if and only if  $f(v_1) \cong_{\mathcal{E}} f(v_2)$  and say  $\cong_{\mathcal{E}}^f$  is the *equivalence induced by  $f$*  on  $N$  under  $\mathcal{E}$ . We may omit  $\mathcal{E}$  if it is irrelevant or clear from the context.

For any function  $f$  we can implement a discriminator for  $\tau$  under  $_f$  as follows:

$$D[(v_1, w_1), \dots, (v_n, w_n)] = \Delta_{\tau'}[(f(v_1), w_1), \dots, (f(v_n), w_n)]$$

Originally, multiset discrimination was formulated as the problem of finding duplicates in an input list. The definition of discriminator the problem of multiset discrimination as partitioning its input under some given equivalence relation (such as equality) was generalized by Cai and Paige [CP95] to allow for discrimination under the equivalence  $\equiv_f$  given by an additional function argument  $f$ : Input elements  $v_1, v_2$  are equivalent,  $v_1 \equiv_f v_2$ , if they are mapped to the same range value under  $f$ ; that is,  $f(v_1) = f(v_2)$ , where  $=$  is the (predefined) equality relation of the result type. The result of the multiset discrimination is then a map from result values to the set (implemented as a list) of values. In other words, in their formulation, a multiset discriminator is a function that computes the inverse of an input function for a given finite input set of domain values.

### 6.2 Bag and set equivalences

#### 6.2.1 Basics extended

We shall now investigate discrimination for the equivalences defined for bags and sets.

To do so, we extend the notions of Section 2 as follows. The construction of values is extended with two alternatives for enumerated finite sets and bags or simply *sets* and *bags*:  $\langle v_1, \dots, v_n \rangle$  and  $\{v_1, \dots, v_n\}$  resulting in

$$v ::= () \mid (v_1, v_2) \mid \text{Inl}(v_1) \mid \text{Inr}(v_2) \mid c \mid n \\ \mid \langle v_1, \dots, v_n \rangle \mid \{v_1, \dots, v_n\}$$

We may also write  $\text{set}[v_1, \dots, v_n]$  and  $\text{bag}[v_1, \dots, v_n]$  to emphasize that, syntactically, sets and bags are but lists annotated with our intention of treating them as sets and bags, respectively.

The size function is extended to simply add the sizes of the elements of a bag or set:

$$\begin{aligned} |()| &= 0 \\ |(v_1, v_2)| &= |v_1| + |v_2| \\ |\text{Inl}(v_1)| &= 1 + |v_1| \\ |\text{Inr}(v_2)| &= 1 + |v_2| \\ |c| &= |c|_a \text{ for } c \in V_a \\ |n| &= 1 \\ |\langle v_1, \dots, v_n \rangle| &= \sum_{i \in \{1..n\}} |v_i| \\ |\{v_1, \dots, v_n\}| &= \sum_{i \in \{1..n\}} |v_i| \end{aligned}$$

We add type constructors  $\text{set}(\cdot)$  and  $\text{bag}(\cdot)$ , which yields

$$\tau ::= 1 \mid \tau \times \tau \mid \tau + \tau \mid a \mid \text{ref}(\tau) \mid \alpha \mid \mu\alpha.\tau \\ \mid \text{bag}(\tau) \mid \text{set}(\tau)$$

The value typing rules of Figure 1 are extended with:

$$\frac{\Gamma \vdash v_1 : \tau \quad \dots \quad \Gamma \vdash v_n : \tau}{\Gamma \vdash \langle v_1, \dots, v_n \rangle : \text{bag}(\tau)}$$

$$\frac{\Gamma \vdash v_1 : \tau \quad \dots \quad \Gamma \vdash v_n : \tau}{\Gamma \vdash \{v_1, \dots, v_n\} : \text{set}(\tau)}$$

Finally, the definition of  $\cong_{\mathcal{E}}$  the congruence on  $V^A[N]$  induced by an equivalence relation  $\mathcal{E}$  on  $N$ , is extended as follows. It is the smallest relation  $R$  closed under the following rules:

1.  $(c, c') \in R$  for all  $c, c' \in V_a$  such that  $c =_a c'$ ;

2.  $((), ()) \in R$ ;
3. if  $(v, v') \in R$  and  $(w, w') \in R$  then
  - (a)  $((v, w), (v', w')) \in R$ ,
  - (b)  $(Inl(v), Inl(v')) \in R$ , and
  - (c)  $(Inr(w), Inr(w')) \in R$ ;
4. if there exists a permutation  $\pi : \{1 \dots n\} \rightarrow \{1 \dots n\}$  such that  $(v_i, w_{\pi(i)}) \in R$  for all  $i$  with  $1 \leq i \leq n$  then  $(\langle v_1, \dots, v_n \rangle, \langle w_1, \dots, w_n \rangle) \in R$ ;
5. if for all  $i \in \{1 \dots m\}$  there exists  $j \in \{1 \dots n\}$  such that  $(v_i, w_j) \in R$  and, conversely, for all  $j \in \{1 \dots n\}$  there exists  $i \in \{1 \dots m\}$  such that  $(v_i, w_j) \in R$  then  $(\{v_1, \dots, v_m\}, \{w_1, \dots, w_n\}) \in R$ .

The last two clauses — the only ones that distinguish bags and sets from ordinary lists — express the bag- and set-properties of bags and sets.

### 6.2.2 Bag discrimination

The basic idea for multiset discrimination of bag values of type  $\langle \tau \rangle$  is as follows: Let  $\leq$  be any total order on the values of  $\tau$ . We can discriminate a list of values

$$[(\langle \vec{v}_1 \rangle, w_1), \dots, (\langle \vec{v}_k \rangle, w_k)]$$

by first sorting the  $\vec{v}_i$  according to *some* total order on their elements and then applying the standard list discriminator to  $[(\text{sort}(<) \vec{v}_1, w_1), \dots, (\text{sort}(<) \vec{v}_k, w_k)]$ .

But how to do this? Many types do not come equipped with a comparison function defining a total order. This holds, in particular, for references.<sup>1</sup> In any case, sorting based on a standard sort operation would require  $\Theta(n \log n)$  applications of the comparison function, which is too slow for us.

We do not need a standard ordering, however: The above statement is correct for *any* total order  $\leq$ . Sorting according to an arbitrary data-dependent ordering, which the algorithm itself finds, has been called *weak-sorting* by Paige [Pai95]:

A function  $f : \text{list}(\text{bag}(\tau)) \rightarrow \text{list}(\text{list}(\tau))$  is a *weakly sorting function* (or *weak-sorter*) if there exists a total order  $\leq$  on  $\tau$  such that

1. for  $f[\vec{v}_1, \dots, \vec{v}_k] = [\vec{v}'_1, \dots, \vec{v}'_l]$  we have  $k = l$  and  $\vec{v}'_i$  is a permutation of  $\vec{v}_i$  for all  $1 \leq i \leq k$ ;
2. each  $\vec{v}'_i$  is sorted according to  $\leq$ ; that is, if  $\vec{v}'_i = [v'_{i1}, \dots, v'_{in_i}]$  then  $v_{ij} \leq v_{i(j+1)}$  for all  $1 \leq j < n_i$ .

---

<sup>1</sup>All hell would break loose if we made a standard comparison function on references available. Just think about implementing garbage collection correctly since it would have to preserve the sort order of references. The absence of an observable standard total ordering on references allows treating stores “up to  $\alpha$ -equivalence”.

Linear-time weak sorters are surprisingly simple to implement by applying multiset discrimination twice. The basic idea is as follows (we are somewhat informal here):

1. Perform multiset discrimination on

$$[(c_{11}, l_1), \dots, (c_{1k_1}, l_1), \dots, (c_{n1}, l_n), \dots, (c_{nk_n}, l_n)];$$

that is, on the *containments* of elements in lists.

2. This results in a permutation of the input such that all the pairs  $(c, l)$  with the same element  $c$  are adjacent to each other. This list establishes a total order on them:  $c < c'$  if all the occurrences of  $c$  occur before all the occurrences of  $c'$ .
3. Flip the components in each pair of the permutation, resulting in a list of pairs  $(l, c)$ .
4. Perform multiset discrimination on that list of pairs, using an order-preserving discriminator.
5. This results in a partition where each equivalence class of contains pairs  $(l, c)$  with the same list  $l$  and with all the elements in  $l$  ordered according to  $<$ , the ordering produced by the first multiset discrimination step, and we are done.

It is easiest to understand this by example. We shall apply weak sorting to a list of strings, with strings viewed as lists of characters. Let the input be [“topside”, “lame”, “male”, “deposit”, “one”].

1. We perform multiset discrimination on  $[(“t”, “topside”), (“o”, “topside”), (“p”, “topside”), \dots, (“n”, “one”), (“e”, “one”)]$ .
2. This results in the partition

$$[ [(“t”, “topside”), (“t”, “deposit”)], [(“o”, “topside”), (“o”, “deposit”), (“o”, “one”)], \dots, [(“m”, “lame”), (“m”, “male”)], [(“n”, “one”)]]$$

and so the permutation

$$[(“t”, “topside”), (“t”, “deposit”), (“o”, “topside”), (“o”, “deposit”), (“o”, “one”), \dots, (“m”, “lame”), (“m”, “male”), (“n”, “one”)]$$

of the input. This establishes the total order

$$“t” < “o” < “p” < “s” < “i” < “d” < “e” < “l” < “a” < “m” < “n”$$

on the letters in the input, which is the order in which they occur with their first occurrences in the input.

3. Flipping the components of the permutation results in

[ (“topside”, “t”), (“deposit”, “t”), (“topside”, “o”), (“deposit”, “o”), (“one”, “o”), . . . , (“lame”, “m”), (“male”, “m”), (“one”, “n”) ].

4. Applying multiset discrimination to this list results in

[ (“topside”, “t”), (“topside”, “o”), . . . ] ]

and, by factoring out the common first component in each block and writing the corresponding list of letters in second component position as a string:

[ (“topside”, “topside”), (“deposit”, “topside”), (“one”, “oen”), (“lame”, “elam”), (“male”, “elam”) ]

Note that each input string is mapped to the unique permutation that satisfies the above total order on the characters.

In this example both discriminators — the first one on characters, the second one on strings — have been order-preserving. For correctness it is necessary and sufficient if only the string discriminator is order-preserving.

**Definition 6.1** [Standard weak sorter] Define  $W_\tau : list(bag(\tau)) \rightarrow list(list(\tau))$  as follows.

Given an input list  $[l_1, \dots, l_n]$  such that

$$l_1 = \langle c_{11}, \dots, c_{1k_1} \rangle, \dots, l_n = \langle c_{n1}, \dots, c_{nk_n} \rangle,$$

compute

$$I = [ (c_{11}, (\hat{l}_1, (\hat{l}_1, c_{11}))), \dots, (c_{1k_1}, (\hat{l}_1, (\hat{l}_1, c_{1k_1}))), \\ \dots \\ (c_{n1}, (\hat{l}_n, (\hat{l}_n, c_{n1}))), \dots, (c_{nk_n}, (\hat{l}_n, (\hat{l}_n, c_{nk_n}))) ]$$

where the  $\hat{l}_i$  are  $n$  distinct references, each pointing the corresponding list  $l_i$ . Then apply  $\Delta_{ref(list(\tau))} \circ concat(\theta) \circ \Delta_\tau$ , the composition of  $\Delta_{ref(list(\tau))}$ , the concatenation operator, and  $\Delta_\tau$  to  $I$  and return the result.  $\square$

**Lemma 6.2 (Standard weak sorter for bags)**  $W_\tau$  is a weak sorter. Furthermore, it runs in time and space  $O(n)$  for inputs of size  $n$ .

By composing the weak sorter with list discrimination we arrive at a discriminator for bags:  $\Delta_{bag(\tau)} = \Delta_{list(\tau)} \circ W_\tau$ .

**Theorem 6.3** There is a multiset discriminator  $\Delta_{bag(\tau)}$  for bags under  $\cong_\mathcal{E}$  that executes in time  $O(n)$  and space  $O(n)$  where  $n$  is the size of the input.

### 6.3 Sorting in linear time

Now that we have seen how to sort in linear time according to *some* total order of the input, a natural question is whether we can “trick” the algorithm to sort according to some predefined order, preferably in linear time.

This is quite simple if it is possible to enumerate all the elements occurring in the actual input in the desired order efficiently.

Consider input  $L = [l_1, \dots, l_n]$  to our weak sorter  $W_\tau$ . We call bag  $l_0$  a *sorting guide* for  $L$  if, for all  $i$  with  $1 \leq i \leq n$ , each element of  $l_i$  also occurs in  $l_0$ . It is a *canonical* sorting guide, if each element occurs at most once in  $l_0$ . A sorting guide defines a total order on its elements:  $c < c'$  if and only if the first occurrence of  $c$  in  $l_0$  occurs before the first occurrence of  $c'$ .

**Corollary 6.4** *Given input  $L = [l_1, \dots, l_k]$  and sorting guide  $l_0$  for  $L$ , applying weak sorter  $W_\tau$  to  $[l_0, l_1, \dots, l_k]$  results in  $[l'_0, l'_1, \dots, l'_k]$  where, for each  $i \in \{1 \dots k\}$ ,  $l'_i$  is a permutation of  $l_i$  sorted according to the sort order defined by  $l_0$ .*

This results in a linear-time sorting algorithm if the sorting guide can be constructed in time  $O(|L|)$ . Note that  $L$  represents  $k$  sorting jobs and not just one, and the cost of constructing the sorting guide may be amortized over all those jobs.

### 6.4 Set discrimination

Weak sorting sorts its input bags into lists where repeated elements (under  $\cong_\mathcal{E}$ ) are placed next to each other. By eliminating adjacent duplicates using equivalence testing under  $\cong_\mathcal{E}$  after (or, more efficiently, during) weak sorting we can eliminate *all* duplicates efficiently. Each element of an input bag enters into at most two equivalence tests. Since each such equivalence test can be decided in time linear in its input, we arrive at weak sorter for *sets*, which sorts its input sets and eliminates all duplicates in the process.

A function  $f : \text{list}(\text{set}(\tau)) \rightarrow \text{list}(\text{list}(\tau))$  is a *weakly sorting function* (or *weak-sorter*) *with duplicate elimination* if there exists a total order  $\leq$  on  $\tau$  such that

1. for  $f[\vec{v}_1, \dots, \vec{v}_k] = [\vec{v}'_1, \dots, \vec{v}'_l]$  we have  $k = l$ , each element of  $\vec{v}_i$  occurs in  $\vec{v}'_i$  and *vice versa*, and  $\vec{v}'_i$  does not contain any duplicates under  $\cong_\mathcal{E}$ ;
2. each  $\vec{v}'_i$  is sorted according to  $\leq$ ; that is, if  $\vec{v}'_i = [v'_{i1}, \dots, v'_{in_i}]$  then  $v_{ij} \leq v_{i(j+1)}$  for all  $1 \leq j < n_i$ .

**Lemma 6.5 (Standard weak sorter for sets)** *There exists a weak sorter  $W'_\tau$  for sets that can be implemented to run in time and space  $O(n)$  for inputs of size  $n$ .*

**Theorem 6.6** *There is a multiset discriminator  $\Delta_{\text{set}(\tau)}$  for sets under  $\cong_\mathcal{E}$  that executes in time and space  $O(n)$  where  $n$  is the size of the input.*

## 6.5 The uniform word problem with associative, commutative and idempotent operators

Consider a binary operator  $\otimes$ . Let  $e, e'$  be expressions (binary trees) built from  $\otimes$  and some given constant values. Let  $\vec{v}, \vec{v}'$  be the leaves of  $e$  and  $e'$ , respectively, in left-to-right order.

- If  $\otimes$  is associative (A) then  $e = e'$  if and only if  $list(\vec{v}) \cong list(\vec{v}')$ .
- If  $\otimes$  is associative and commutative (AC) then  $e = e'$  if and only if  $bag(\vec{v}) \cong bag(\vec{v}')$ .
- If  $\otimes$  is associative, commutative and idempotent (ACI) then  $e = e'$  if and only if  $set(\vec{v}) \cong set(\vec{v}')$ .

Multiset discrimination for bags and sets provides an asymptotically optimal method for deciding the uniform word problem for any number of operators that are associative, commutative, idempotent, or any combination thereof. This is done by converting each subtree with the same associative operator  $\otimes$  into a right-skewed tree whose left child is *not* an  $\otimes$ -node and then classifying it as a list (if  $\otimes$  is only associative), a bag (if  $\otimes$  is associative and commutative) or set (if  $\otimes$  is associative, commutative and idempotent). Performing multiset discrimination of two thus normalized expressions decides whether they are algebraically equal or not. More generally, multiset discrimination of more than two expressions decides all pairwise equalities in one go.

**Theorem 6.7** *Uniform word problem in linear time.*

## 7 Multidiscrimination under isomorphism

Standard discrimination discriminates values in linear time under  $\cong_{\mathcal{E}}$  for some given equivalence relation on the nodes of the global store. It discriminates nodes under the given equivalence relation, not the isomorphism relation  $\cong$ . This basically means that we can discriminate data structures that are represented as proper trees, with no sharing of internal nodes in such trees.

Extending standard discrimination naively to data structures with shared nodes does not work (efficiently). Consider the binary tree  $t_n$  where  $t_0 = Leaf(0)$  and  $t_{i+1} = Node(t_i, t_i)$  for  $i \geq 0$ . Its (tree) size is  $2^{n+2} - 2$ , even though it can be represented by an ordered directed acyclic graph (dag) with  $n + 1$  nodes and  $2n$  edges. Our standard discriminator, applied to the list  $[(t_n, 0), (t_n, 1)]$ , will take  $\Theta(2^n)$  time even though the input can be presented by a data structure of size  $O(n)$ .

Let  $\cong$  be any equivalence relation with the property that if  $E(n) = E(n')$  then  $n \cong n'$ . Define the directed graph  $G = ([V]_{\cong}, E)$  as the least graph on the equivalence classes of  $\cong$  satisfying: if  $(n, n') \in E$  then  $([n]_{\cong}, [n']_{\cong}) \in [E]_{\cong}$ .

**Definition 7.1** [Dagified Store] Store  $S = (N, E)$  is *dagified* by  $\mathcal{E}$  if we have for all  $n, n' \in N : n \cong_S n'$  if and only if  $(n, n') \in \mathcal{E}$ .  $\square$

With equivalence relation  $\mathcal{E}$  in hand, we can easily contract replace each node by a unique ee each A dagified heap has a number of important advantages over an equivalent undagified heap:

1. It enables contracting the store by replacing equivalent nodes by unique equivalence class representative resulting in a store where no two nodes are isomorphic.
2. It admits a constant time equivalence test under  $\cong_S$ .
3. It admits discrimination under  $\cong_S$  by simply discriminating nodes under  $\mathcal{E}$ .

Note that fast discrimination (advantage 3) subsumes fast equality testing (advantage 2).

The problem is producing dagified heaps efficiently. If a heap is required to be in dagified form at all times, the box operation can no longer be implemented as a simple-minded allocation of a new node in the heap since the value may already be stored. Classically, checking whether a value is already stored is done by *value numbering*, also called *hashed consing*, where a mapping from values already stored in a heap to the boxes mapping to them is maintained.

This mapping is usually implemented as a hash table. the argument of a box operation is first hashed and then looked up in the hash table. If the value already exists in it, this means the value is already stored in the heap and the corresponding node is returned. If it does not yet exist, the value is allocated on the heap with a new node and the mapping from the value to the new node is added to the hash table. In essence the hash table  $T$  represents an inverse of the heap  $H = (R, V, E)$ : for each  $b \mapsto v \in E$  there is  $v \mapsto b \in T$ , and *vice versa*. Hashed consing works well in some situations, but has its disadvantages: At least twice as much storage is consumed (in practice much more) compared to simple allocation, if all the box operations happen to be applied to distinct values. Also, the lookup in the hash table adds substantial overhead, which slows down the box operation typically by at least one order of magnitude in comparison to a simple allocation on the heap.

It is rarely required or even advantageous for a heap to be kept in dagified form *at all times*. Note that it is sufficient for fast discrimination that it be in dagified form *just before* discrimination is applied.

We shall now show, how an *acyclic* heap can be dagified in time  $O(m)$  and and a *cyclic* store in time  $O(m \log n)$  time, without maintaining a sizeable auxiliary data structure such as a hash table.

## 7.1 Dagification for acyclic stores

**Definition 7.2** [Induced graph] Let  $S = (N, E)$  be a store. The directed graph  $[S]_{\sim}$  induced by some equivalence relation  $\sim$  on  $V$  is defined as  $([N]_{\sim}, [E]_{\sim})$  where:

1.  $[N]_{\sim}$  are the equivalence classes of  $N$  under  $\sim$ ;
2.  $([n_1], [n_2]) \in [E]_{\sim}$  if and only if there exist  $n'_1, n'_2 \in N$  such that  $n_1 \sim n'_1$ ,  $n_2 \sim n'_2$  and  $n'_2$  occurs in  $E(n'_1)$ .

We write  $[S]$  for the graph induced by  $\perp_N$ . We say  $S$  is *acyclic under*  $\sim$  if  $[S]_{\sim}$  is acyclic. We say  $S$  is *acyclic*, if  $[S]$  is acyclic.  $\square$

Note that if  $S$  is cyclic then it is cyclic under any equivalence relation. The converse is not universally true, of course: there are equivalence relations that make  $[S]_{\sim}$  cyclic without  $S$  being cyclic. As we shall see, we have, however, a weak converse: If  $[S]$  is acyclic then so is  $[S]_{\cong_S}$  and even some coarser equivalences preserve acyclicity. We can exploit this by performing dagification in a two-step process: First find an easily computed equivalence relation  $\sim$  that contains  $\cong_S$  and preserves acyclicity of  $S$ . Then process the  $[N]_{\sim}$ -equivalence classes in reverse topological order.

### 7.1.1 Staging

**Definition 7.3** [Staging equivalence] Let  $S = (N, E)$  be a store. We say that equivalence relation  $\sim$  on  $N$  is a *staging equivalence* for  $\cong$ , the isomorphism relation on  $N$ , if:

1.  $\sim$  is equal to or coarser than  $\cong$ ; that is, for all  $n, n' \in N$ , if  $n \cong n'$  then  $n \sim n'$ ;
2.  $[S]_{\sim}$  is acyclic.

$\square$

The first property guarantees that we can dagify each block independently of any other block since nodes from different  $\sim$ -blocks cannot be isomorphic. The second property guarantees that we can do the discrimination efficiently in stages, by reverse topological ordering the  $\sim$ -classes according to  $[H]_{\cong}$ .

The standard staging relation used so far is based on the observation that nodes at different heights in a tree cannot be isomorphic.

**Lemma 7.4 (Depth staging)** *Let  $S = (N, E)$  be an acyclic store. Define  $\text{depth}(S)n$  to be the length (measured as the number of edges traversed) of a longest path in  $[S]$  originating from  $n$ . Then  $n \sim_D n'$  if and only if  $\text{depth}(S)n = \text{depth}(S)n'$  is a staging equivalence for  $\cong$ .*

Define  $zip([v_1, \dots, v_k], [w_1, \dots, w_k]) = [(v_1, w_1), \dots, (v_k, w_k)]$ . Let  $S = (N, E)$  be an acyclic store.

1. Compute a staging relation and order its equivalence classes in reverse topological order  $[B_1, \dots, B_k]$ .
2. Initialize  $\mathcal{E} := \perp_N$ .
3. For  $i = 1 \dots k$  do  $\mathcal{E} := \mathcal{E} \vee \Delta(zip(B_i, B_i))$ ;

Figure 3: Acyclic Dagification algorithm. Note that  $\Delta(zip(B_i, B_i))$  discriminates  $B_i$  under  $\mathcal{E}$ .

PROOF Let  $R$  be an equivalence relation such that  $R = \cong_R$ . By induction on the definition of  $v \cong_R v'$  we can show that if  $v \cong_R v'$  then for all  $n \in N$  occurring in  $v$  there exists an  $n' \in N$  occurring in  $v'$  such that  $(n, n') \in R$ .

From this we can conclude that if  $n \cong_R n'$  then  $depth(S)n = depth(S)n'$ . Assume otherwise that  $k = depth(S)n > depth(S)n'$ . Let  $n = n_0 \rightarrow \dots \rightarrow n_k$  a longest path in  $[E]$ . By the previous observation there must be a path  $n' = n'_0 \rightarrow \dots \rightarrow n'_k$  such that  $n_i \cong_R n'_i$  for all  $1 \leq i \leq k$ . This shows that  $depth(S)n' \geq k$ , which is in contradiction to our assumption that  $depth(S)n' < k$ . This shows that  $n \sim_D n'$  implies  $n \cong_R n'$ .  $\square$

Other staging relations are possible and indeed necessary if the closure conditions are extended with *ground axioms* relating nodes at different heights. See Downey, Sethi, Tarjan [DST80] for an example of this.

With a staging relation as a guide we can dagify an acyclic store in bottom-up fashion in a single pass, as shown in Figure 3.

**Lemma 7.5** *Let  $R$  be any relation such that  $R = Equiv_S(R)$ . Then after the  $i$ -th iteration of the for-loop in Figure 3 the following holds:  $R \upharpoonright_{B_1 \cup \dots \cup B_i} = \mathcal{E}$ .*

PROOF By induction on  $i$ .  $\square$

This lemma shows that  $Equiv_S(\cdot)$  has a unique fixed point, which we can compute “bottom-up”, basically by a least fixed point computation even though  $\cong$  is defined as a greatest fixed point.

**Theorem 7.6** *Let  $S = (N, E)$  be a acyclic store, then  $S$  can be dagified in time and space  $O(n)$  where  $n = |S|$ .*

PROOF The depth staging can be computed and topologically sorted in  $O(n)$  time. The algorithm in Figure 3 also executes in time  $O(n)$ . Since  $\cong_S = Equiv_S(\cong_S)$ , it computes  $\mathcal{E}$  such that  $\mathcal{E} = \cong_S$ .  $\square$

Once we have computed an equivalence relation that dagifies an input graph, we can employ standard multiset discrimination under  $\mathcal{E}$  to obtain multiset discrimination under isomorphism.

**Corollary 7.7 (Optimal discrimination under isomorphism for acyclic stores)**

*For any  $\tau$ , there is a multiset discriminator  $\Delta_{\tau}^{\text{mathit{iso}}}$  for  $\tau$  under isomorphism which executes on store  $S$  and with input  $R$  in time  $O(n)$  where  $n = |S| + |R|$ .*

## 8 Multiset discrimination for cyclic structures

We have seen how to perform multiset discrimination under isomorphism on arbitrary trees (without sharing) and dags (with sharing), even under the rich equivalences induced by bag- and set-values. In all cases, we could perform multiset discrimination in linear time using only pointer machine operations.

The key step in handling sharing (of nodes) efficiently is dagification. This was accomplished for acyclic stores by staging the processing of nodes in This is clearly not possible for cyclic stores. In this section we shall show how dagification for cyclic stores can be accomplished in time  $O(m \log n)$  and space  $O(m)$  for store  $S = (N, E)$  where  $m = |S|$  and  $n = |N|$ . This is the first time we have to resort to a superlinear complexity bound. It holds for arbitrary stores, with cycles and sharing, with set- and bag-values and the associative-commutative-idempotent properties they embody. This generalizes the above-mentioned algorithms and yet matches their asymptotic performance. There is but one subproblem that is known to perform better for cyclic graphs (stores): the single-function coarsest partition problem, which, in our terminology, corresponds to dagification of stores where all nodes are mapped to tagged singleton sets (or lists or bags for that matter) of nodes; that is,  $E(n)$  has the form  $(c, [n'])$  for  $c \in \overline{K}$  for some  $K$ . That problem can be solved in linear time [PT84, PTB85].

This section draws on and generalizes three algorithms from the literature and shows that they solve related problems in our terminology: Dagification for cyclic graphs where the neighbors of each node form a tagged list, bag, or set, respectively. Downey, Sethi and Tarjan solved the problem for lists, calling it the common subexpression problem [DST80]; Cardon and Crochemore solved it for bags, referring to it as finding the coarsest regular congruence refining an initial partition [AM82]; and Paige and Tarjan solved it for sets, calling it the relational coarsest partition problem [PT87, Section 3]. (Paige and Tarjan also gave a simpler algorithm than Cardon and Crochemore's for bags, calling it the size-stable coarsest partition refinement problem.)

Each of these algorithms runs in time  $O(m \log n)$  where  $n$  is the number of nodes and  $m$  the number of (multi)edges.

The algorithms have a common algorithmic structure: formulate the problem as a (greatest) fixed point problem and then compute the result by dominated convergence [CP89] to enable efficient incremental computation in each iteration step. The key step, solved differently in each case, is efficient incremental computation of the necessary updates by employing a “modify-the-smaller-half” principle, which was introduced by Hopcroft for the minimization of deterministic finite automata [Hop71].

We shall see that, using the transformational algorithmic ideas of [CP89], the algorithms can be combined into a single framework that allows dagification of arbitrary stores; that is, stores where each node may be mapped to any value whatsoever, amongst which lists, bags, and sets as special cases.

As a corollary we obtain that any store of size  $m$  and with  $n$  nodes can be dagified in time  $O(m \log n)$ , enabling multiset discrimination pursuant to this time  $O(n)$ .

In the following we shall assume, without loss of generality, that our stores are fully boxed; that is, the only values stored in it are of the form

$$v ::= () \mid (n_1, n_2) \mid Inl(n_1) \mid Inr(n_2) \mid c.$$

## 8.1 Greatest fixed point computation

Recall that  $\cong_S$  for  $S = (N, E)$  is the largest congruence relation  $R$  such that  $\forall n, n' \in N : (n, n') \in R \Leftrightarrow E(n) \cong_R E(n')$ ; that is, it is the maximal fixpoint of  $Equiv_S(\cdot)$ . We can compute the maximal fixed point by Kleene induction, starting with the maximal partition  $\mathcal{E}_1 = \top_N$  as its only block, and then computing  $\mathcal{E}_{i+1} = Equiv_S(\mathcal{E}_i)$  until a fixed point is reached. A single iteration step can be performed in  $O(m)$  time by multiset discrimination of  $[(E(n), n) : n \in V]$  under  $\cong_{\mathcal{E}_i}$ , where  $m = |S|$ . The result of this is  $\mathcal{E}_{i+1}$ . Since there can be at most  $n$  such iteration steps this immediately yields an  $O(nm)$  algorithm.

We can improve upon this. The first step is computing the fixed point by *dominated convergence* [CP89], which prepares for efficient incremental computation of each iteration step, as shown in Figure 4.

The bottleneck is Step 3 since a naive implementation of this steps requires time  $O(m)$ . Note that Figure 4 maintains  $\mathcal{F}$  as a refinement of  $\mathcal{E}$ ; that is, each block of  $\mathcal{E}$  is made up of one or more blocks of  $\mathcal{F}$ . We can construct  $\mathcal{E}'$  in Step 2 of Figure 4 by choosing a block (equivalence class)  $B$  of  $\mathcal{E}$  such that  $B = B_1 \uplus \dots \uplus B_k$  for  $k \geq 2$  for blocks  $B_1, \dots, B_k$  of  $\mathcal{F}$  and defining  $\mathcal{E}'$  to be  $\mathcal{E}$  where block  $B$  is split into two blocks:  $B_1$  and  $B_2 \uplus \dots \uplus B_k$ .

Now consider  $N' = [E]^{-1}B_1$ , the set of nodes  $\{n \in N \mid n' \in B_1 \wedge (n, n') \in [E]\}$ , and their complement  $N'' = N - N'$ . It is easy to see that for all  $n, n' \in N''$  we have  $(n, n') \in Equiv_S(\cdot)(\mathcal{E}')$  if and only if  $(n, n') \in Equiv_S(\cdot)(\mathcal{E})$  since  $\mathcal{E} = \mathcal{E}'$  when restricted to  $N - B_1$ . Furthermore, if  $(n, n') \in Equiv_S(\cdot)(\mathcal{E}')$  then both  $n, n'$  belong to  $N'$  or they both belong to  $N''$ . This gives us

Let  $\mathcal{E} := \top_N$  and  $\mathcal{F} := \text{Equiv}_S(\mathcal{E})$ .

The following loop maintains the invariant  $\mathcal{F} = \text{Equiv}_S(\mathcal{E})$ .

1. If  $\mathcal{F} = \mathcal{E}$  terminate. The answer is  $\mathcal{E}$ . Otherwise continue with the next step.
2. Find  $\mathcal{E}'$  such that  $\mathcal{F} \leq \mathcal{E}' < \mathcal{E}$ .
3. Compute  $\mathcal{F}' := \text{Equiv}_S(\mathcal{E}')$ .
4. Set  $(\mathcal{E}, \mathcal{F}) := (\mathcal{E}', \mathcal{F}')$  and continue with Step 1

Figure 4: Iterative computation by dominated convergence

the following lemma, which expresses that we can compute  $\mathcal{F}'$  in Step 3 of Figure 4 incrementally by only processing elements of  $N'$ , the nodes that depend on  $B_1$ .

**Lemma 8.1 (Distributivity of multiset discrimination)**  *$\text{Equiv}_S(\cdot)(\mathcal{E}')$  can be computed as follows:  $\mathcal{F}' := \mathcal{F} \wedge \text{Equiv}_S(\mathcal{E}' \upharpoonright_{N'})$  where  $\mathcal{E}' \upharpoonright_{N'}$  denotes  $\mathcal{E}'$  restricted to the set  $N'$ .*

Intuitively, think of the splitting step as assigning the nodes in  $B_1$  a new  $\mathcal{E}$ -equivalence class pointer. All the other nodes retain their equivalence class pointers and so multiset discrimination on those nodes whose values contain only elements of  $N - B_1$  just gives the same result as before  $B_1$  was split off.

If the store does not contain any set- or bag-values it is easy to see that the number of edges of  $[E]$  incident to  $N'$  is linearly bounded by the number of edges incident to  $B_1$ .

We can always choose  $B_1$  such that  $|B_1| \leq |B|/2$ . This means that any node  $n \in N$  is an element of a  $B_1$  chosen in the split step at most  $O(\log |N|)$  times. This “modify-the-smaller-half” strategy yields:

**Theorem 8.2** *Store  $S = (N, E)$  without set- or bag-values can be dagified in  $O(|E| \log |N|)$  time and  $O(|E|)$  space.*

This is basically Downey, Sethi, Tarjan’s result for cyclic graphs [DST80], generalized to apply to 1 and sum type values. Note, however, that our use of standard multiset discrimination in the incremental computation step of  $\mathcal{F}'$  yields a somewhat better combined complexity bound than the variants they give.

## 8.2 Incremental discrimination for bags

The above theorem does not immediately extend to stores with bag-values since bag-values have an outdegree that is not bounded by a constant. In the worst case, each of the recomputation steps according to Lemma 8.1 may take time  $\Theta(m)$  time resulting in  $\Theta(mn)$  complexity even when applying the modify-the-smaller-half principle.

We can reduce the complexity as follows. Let  $\mathcal{E}'$  be constructed from  $\mathcal{E}$  by splitting off block  $\mathcal{B}_\infty$ , as above. Consider a block  $C$  of bag-valued nodes. Let  $n, n'$  be two nodes in  $C$  with  $E(n) = \langle n_1, \dots, n_k \rangle$  and  $E(n') = \langle n'_1, \dots, n'_k \rangle$ . Now, we know that  $E(n)$  and  $E(n')$  have equally many elements from each  $\mathcal{E}$ -block, and in particular from block  $B$ . Assume they both have  $k_B$  elements of  $B$ . After splitting  $B$  into  $B_1$  and  $B - B_1$  when forming  $\mathcal{E}'$ , let  $E(n)$  and  $E(n')$  have  $k_{B_1}$ , respectively  $k'_{B_1}$  elements of  $B_1$ . Note that they then have  $k_B - k_{B_1}$ , respectively  $k_B - k'_{B_1}$  elements of  $B - B_1$  and both still have the same number of elements from any other block of  $\mathcal{E}'$ . It follows that  $E(n) \cong_{\mathcal{E}'} E(n')$  and thus  $(n, n') \in \text{Equiv}_S(\mathcal{E}')$  if and only if  $k_{B_1} = k'_{B_1}$ . In other words, we can avoid visiting any of the nodes outside  $B_1$ . More specifically, during the recomputation step of Lemma 8.1 replace  $S = (N, E)$  by  $S' = (N, E')$  where  $E'(n) = \langle n_1, \dots, n_k \rangle$  if  $E(n)$  is bag-valued and  $n_1, \dots, n_k$  are all the nodes in  $E(n)$  that are elements of  $B_1$ . Now the number of edges incident to  $N'$  is again linearly bounded by the number of edges incident to  $B_1$ .

**Theorem 8.3** *Store  $S = (N, E)$  without set-values can be dagified in  $O(|E| \log |N|)$  time and  $O(|E|)$  space.*

Since  $E'(n)$  contains only elements of a  $B_1$ , a single  $\mathcal{E}'$ -block, it is sufficient to calculate the number of  $B_1$ -elements of  $E'(n)$  and replacing multiset discrimination for bag-valued nodes by multiset discrimination on numbers.

This theorem when restricted to stores that contain bag-values only, has been proved by Cardon and Crochemore [AM82]. Paige and Tarjan [PT87] presented a somewhat simplified algorithm with the same bounds. The above theorem combines

## 8.3 Incremental discrimination for sets

Dagification for set-values is the most difficult dagification subclass. (See Paige, Tarjan [PT87] for a discussion as to why.)

Let  $\mathcal{E}'$ , as before, be constructed from  $\mathcal{E}$  by splitting off block  $\mathcal{B}_\infty$ . Consider two nodes  $n, n'$  in a  $\mathcal{E}$ -block  $C$  of set-valued nodes.

in  $C$  with  $E(n) = \langle n_1, \dots, n_k \rangle$  and  $E(n') = \langle n'_1, \dots, n'_k \rangle$ .

Now, we know that  $E(n)$  and  $E(n')$  have elements of the same  $\mathcal{E}$ -blocks. If  $B$  is not amongst those blocks, splitting off  $B_1$  will not change anything and  $n$  and  $n'$  will also be in the same block in  $\mathcal{F}' = \text{Equiv}_S(\mathcal{E}')$ . Let us

assume then  $E(n)$  and  $E(n')$  both contain elements of  $B$ .  $n$  can be classified into one of the three categories: categories:

1. all  $B$ -nodes of  $E(n)$  are in  $B_1$ ;
2. none of the  $B$ -nodes of  $E(n)$  are in  $B_1$ ;
3. at least one  $B$ -node of  $E(n)$  is in  $B_1$  and at least one  $B$ -node of  $E(n)$  is in  $B - B_1$ .

The same can be done for  $n'$ . After splitting  $B_1$  off  $B$  to define  $\mathcal{E}'$  it is the case that  $E(n) \cong_{calE'} n'$  (and thus  $(n, n') \in \mathcal{F}'$ ) if and only if  $n$  and  $n'$  fall into the same category. Given that we have a list of nodes  $B_1$  the hard part is distinguishing between cases 1 and 3 without visiting any of the nodes in  $N - B_1$ . Paige and Tarjan [PT87] solve this problem by maintaining for each node a count of how many elements of each  $\mathcal{E}$ -block it has. When splitting  $B_1$  off the count for  $B$  of each node  $n$  with an element in  $B_1$  is decremented by 1. After all nodes in  $B_1$  are processed the thus decremented count for  $B$  represents the count for  $B - B_1$ . Now we can distinguish the two cases: Node  $n$  falls into category 1 if the count for  $B - B_1$  is zero; if it is nonzero it falls into category 3.

Implementing this efficiently is nontrivial since it requires constant-time access to the count for any block of any node. We refer to [PT87, Section 3] for details.

Note that now we can bound the number of edges incident to  $N'$  again linearly by the number of edges incident to  $B_1$ . Combining this with the treatment of bags and other values, we arrive at the main dagification theorem.

**Theorem 8.4** *Store  $S = (N, E)$  can be dagified in  $O(|E| \log |N|)$  time and  $O(|E|)$  space.*

#### 8.4 General multiset discrimination

We can construct a discriminator, as before, by composing it from a dagification step followed by standard multiset discrimination under the dagifying equivalence relation. This yields the following general corollary.

**Corollary 8.5** *Let  $S = (N, E)$  be a store. For any  $\tau$ , there exists a discriminator  $\Delta_\tau^{iso}$  for  $\tau$  under isomorphism which executes on store  $S$  and with input  $R$  in time  $O(m \log n)$  and space  $O(m)$  where  $m = |S| + |R|$  and  $n = |N|$ .*

## 9 Applications

## 10 Experiments

## 11 Conclusion

## Acknowledgments

Bob Paige and his co-authors invented all the algorithmic ideas and techniques behind multiset discrimination over a series of papers, including standard multiset discrimination, weak sorting and coarsest partitioning in cyclic graphs. He would be pleased to find that they can be combined into a general framework built around our notion of a discriminator, but would hardly be surprised.

Ken Friis Larsen has graciously helped with the blind testing.

Olivier Danvy encouraged me to write about multiset discrimination, focusing on its general applicability as a practical algorithmic tool rather than the specific algorithmic uses it has been put to.

## References

- [AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [AM82] Cardon A. and Crochemore M. Partitioning a graph in  $o(|a| \log_2 |v|)$ . *Theoretical Computer Science (TCS)*, 19:85–98, 1982.
- [CP89] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1989.
- [CP95] Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science (TCS)*, 145(1-2):189–228, July 1995.
- [DST80] P. Downey, R. Sethi, and R. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.
- [Hop71] John Hopcroft. An  $n \log n$  algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.
- [JJ96] Johan Jeuring and Patrik Jansson. Polytypic programming. In *Advanced Functional Programming*, Lecture Notes in Computer Science, pages 68–114. Springer-Verlag, 1996.

- [Meh84] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume I of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1984.
- [Pai95] Robert Paige. Analysis and transformation of set-theoretic languages. Technical Report NS-95-5, BRICS, Dept. of Computer Science, University of Aarhus, Notes on mini-course held August 14-17, 1995 1995. ISSN 0909-3206.
- [PT84] R. Paige and R. Tarjan. A linear time algorithm to solve the single function coarsest partition problem. In *Proc. Int'l Conf. on Algorithms, Languages, and Programming*, 1984.
- [PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.
- [PTB85] R. Paige, R. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science*, 40:67–84, 1985.
- [Rey83] J. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, pages 513–523, 1983.
- [Tar83] R. Tarjan. *Data Structures and Network Flow Algorithms*, volume CMBS 44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.
- [Wad89] P. Wadler. Theorems for free! In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 347–359. ACM Press, September 1989.
- [ZGC03] Yoav Zibin, Joseph Gil, and Jeffrey Considine. Efficient algorithms for isomorphisms of simple types. In *Proc. 2003 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 160–171. ACM, ACM Press, January 2003. SIGPLAN Notices, Vol. 38, No. 1.