

Generic Discrimination

Sorting and Partitioning Unshared Data in Linear Time^{*}

Fritz Henglein

Department of Computer Science, University of Copenhagen (DIKU)
henglein@diku.dk

Abstract

We introduce the notion of *discrimination* as a generalization of both sorting and partitioning and show that worst-case linear-time discrimination functions (discriminators) can be defined *generically*, by (co-)induction on an expressive language of *order denotations*. The generic definition yields discriminators that generalize both distributive sorting and multiset discrimination. The generic discriminator can be coded compactly using list comprehensions, with order denotations specified using Generalized Algebraic Data Types (GADTs). A GADT-free combinator formulation of discriminators is also given.

We give some examples of the uses of discriminators, including a new most-significant-digit lexicographic sorting algorithm.

Discriminators generalize binary comparison functions: They operate on n arguments at a time, but do not expose more information than the underlying equivalence, respectively ordering relation on the arguments. We argue that primitive types with equality (such as references in ML) and ordered types (such as the machine integer type), should expose their equality, respectively standard ordering relation, as discriminators: Having *only* a binary equality test on a type requires $\Theta(n^2)$ time to find all the occurrences of an element in a list of length n , for each element in the list, even if the equality test takes only constant time. A discriminator accomplishes this in linear time. Likewise, having only a (constant-time) comparison function requires $\Theta(n \log n)$ time to sort a list of n elements. A discriminator can do this in linear time.

Categories and Subject Descriptors D [1]: 1; F [2]: 2

General Terms Algorithms, Languages, Theory

Keywords discrimination, discriminator, equivalence, functional, generic, multiset discrimination, order, partitioning, sorting, total preorder

1. Introduction

Sorting has numerous applications and is one of the most fundamental problems in computer science: There is probably no text

^{*}This work has been partially supported by the Danish Research Council for Nature and Universe (FNU) under the grant *Applications and Principles of Programming Languages (APPL)*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-59593-919-7/08/09...\$5.00

book on algorithms that does not cover sorting in one of its first chapters.

A related problem is partitioning: Given an equivalence relation, partition a set of inputs into its equivalence classes. Often sorting is used as a subsidiary step to partitioning: Given a total preorder¹ whose induced equivalence is the desired equivalence relation, sort the input and then group *runs* of equivalent elements together.

1.1 Discrimination, sorting, partitioning

In this paper we develop the notion of *discrimination* as a generalization of both sorting and partitioning.

A *discriminator* (*discrimination function*) is a function that takes a list of key-value pairs as input and returns the values in groups according to their keys: values are grouped together if and only if they are associated with equivalent keys in the input according to some given equivalence relation on keys. It is *stable* if the values in each group occur in the same relative order as in the input.

A *sorting discriminator* is a discriminator that returns the groups of values consistent with a given ordering relation on the keys. For example, the stable sorting discriminator for the standard order on natural numbers maps $[(4, "foo"), (8, "bar"), (4, "baz")]$ to $[[["foo", "baz"], ["bar"]]]$.

Sorting functions, (sorting) partition functions and (sorting) discriminators can be thought of as variations of each other. They have slightly different types: The output of a sorting function is a list of key-value pairs, just like its input. A partition function returns runs of key-equivalent pairs as explicit lists (groups): it returns a list of lists of key-value pairs. A discriminator does the same, but drops the key components and returns only the value components: it returns a list of lists of values.

Even though a discriminator drops the keys in its output, we can use it to sort. Preprocess the input by associating each pair with its key: $[(4, (4, "foo")), (8, (8, "bar")), (4, (4, "baz"))]$. Apply the discriminator, which yields $[[[(4, "foo"), (4, "baz")], [(8, "bar")]]]$ —this is actually the output of a partitioner—and finally flatten, resulting in $[(4, "foo"), (4, "baz"), (8, "bar")]$. Discriminators are parametrically polymorphic in their value part: Once the keys have been extracted in the preprocessing stage, only pointers to the original key-value pairs need to be passed to the discriminator as value components in the input; they are not dereferenced during discrimination.

A *generic (sorting) discriminator* is a function that maps specifications of equivalence relations (total preorders) to discriminators for the denoted equivalence relations (total preorders). To warrant

¹A total preorder is a binary relation R that is transitive and total, but not necessarily antisymmetric.

the use of “generic”² our domain of specifications is rather expressive. It includes a potentially unlimited number of ordering relations on all first-order regular recursive types.

What makes discriminators the central notion of this paper is that they turn out to be the “natural” notion for an inductive definition, from which the other notions, sorting and partition functions, can be defined parametrically. This is, loosely put, analogous to discriminators playing the role of an inductively proved lemma from which sorting and partitioning functions (noninductively) follow as “theorems”.

1.2 Contributions

Our contributions are:

- An expressive language for denoting total preorders (orders).
- Purely functional generic definitions of comparison functions, sorting functions and discriminators over order denotations,
- Sorting, partition and discrimination functions that execute in worst-case linear time without compromising abstraction: only the ordering relation is observable (no coding into the integers).
- Complete specification of generic discrimination in less than 30 lines of Haskell, employing list comprehension and Generalized Algebraic Data Types (GADTs).
- Illustrations of how algorithms for specific equivalences can be coded by simply specifying the equivalence relation in question and applying the generic discriminator to it.
- A discussion of the asymptotic time complexity of sorting since it seems to be subject to some popular confusion.
- The conclusion that ordered types, in particular primitive types in programming languages, should make their ordering relation available by way of a discriminator or sorting function, not only a comparison function; analogously, but with even more importance, that types with equality should expose their equality by an discriminator, not only a binary equality test.

1.3 Overview

After some prerequisites (Section 2) we introduce a language for denoting total preorders (Section 3). This sets the stage for defining a number of generic functions on such order denotations, culminating with a generic stable sorting discriminator (Section 4). We then analyze which order denotations give rise to linear-time discriminators (Section 5). We demonstrate the use of the generic discriminator and its variants (Section 6). Finally we exhibit a discriminator combinator library derived from the generic discriminator (Section 7), discuss various aspects of discrimination, including the complexity of sorting (Section 8), and offer some conclusions (Section 9).

Generic discrimination for equivalence denotations is analogous to discrimination for orders. For reasons of space it is not explicitly covered here, however.

2. Prerequisites

2.1 Basic mathematical notions

A *total preorder* (S, R) is a set S together with a binary relation $R \subseteq S \times S$ that is transitive, total³ and thus also reflexive. We call R an *ordering relation* with domain S . A total preorder is a *total order*

²There is no technical definition of the term “generic”. It is broadly applied to explicit parametric polymorphism, as in CLU, Ada, Java, C#; or template programming as in C++; or intensional polymorphism/polytypic programming as in PolyP. Our use is in the style of the latter.

³ $\forall x, y \in S : (x, y) \in R \vee (y, x) \in R$

(without the “pre”) if it is anti-symmetric. What we call *orders* henceforth are total preorders: anti-symmetry is not required.

We say (S, R) is an *order on* S' if (S, R) is an order and $S \subseteq S'$.

An *equivalence* (S, E) is a set S together with a binary relation $E \subseteq S \times S$ that is reflexive, symmetric and transitive. We call E an *equivalence relation*.

Each ordering relation R canonically induces an equivalence relation: $x \equiv_R y \Leftrightarrow R(x, y) \wedge R(y, x)$.

2.2 Haskell notation

To specify concepts and simultaneously provide an implementation for ready experimentation we use Haskell notation. Specifically, we frequently employ list abstractions for convenience and, as we hope, readability.⁴

Order denotations are formulated using Generalized Algebraic Data Types (GADTs), as implemented in the Glasgow Haskell Compiler (Glasgow Haskell), for immediate execution. Just as the Haskell notation itself they should be thought of as convenient term representations. As we point out, they can be eliminated, yielding a combinator implementation in standard Haskell 98.

3. Order denotations

An *ordered type* is a type denoting a set with a designated ordering relation. In this paper our types are restricted to some given primitive types, plus products, sums and regular recursive types interpreted inductively.

Types often come with implied standard ordering relations: the standard order on natural numbers, the ordering on character sets given by their numeric codes, the lexicographic (alphabetic) ordering on strings over such character sets, and so on.

We quickly discover the need for more than one ordering relation on a given type, however: descending instead of ascending order; ordering strings by their first 4 characters and ignoring the case of letters, etc. For this purpose we introduce a typed language of *order denotations*; that is, terms that denote ordering relations. We write `Order T` for the type of denotations of orders on `T`.

3.1 Primitive order denotations

We assume that each primitive ordered type comes equipped with a standard denotation for its ordering relation. Examples are 8-bit characters under their canonical Latin1-ordering, machine integers under their standard numeric ordering and the one-element type with its trivial order. Without loss of generality we restrict ourselves to ordering relations on finite segments $0 \dots n - 1$ here. Unsigned machine integers can be thought of as products of bytes and unbounded integers as lists of bytes. We shall see how to define efficient discriminators for such composed types from their component types.

```
-- Char : ascending alphabetic order on 8-bit characters
Char    :: Order Char
-- Nat n : standard ascending order on {0, ..., n-1}
Nat     :: Int -> Order Int
-- Unit : trivial order on ()
Unit    :: Order ()
```

3.2 Order constructors

Given orders on types we can define canonical orders on product and sum types: Pairs are ordered according to the first component, with resulting equivalences resolved by ordering according to the second components; elements of a sum type are ordered such that left-tagged elements come first, followed by right-tagged elements,

⁴Some readers may prefer a combinatory “point-free” notation, though.

with equally-tagged elements resolved by the given summand orders.

```
-- Sum r1 r2 : order on tagged values, left elements first,
--             left elements ordered according to r1,
--             right elements according to r2;
Sum      :: Order t1 → Order t2 → Order (Either t1 t2)
-- Pair r1 r2 : lexicographic order on pairs,
--             ordered by first components according to r1,
--             equivalent first components ordered
--             by second components according to r2
Pair     :: Order t1 → Order t2 → Order (t1, t2)
```

Given an ordering relation R_2 on the co-domain of a function $f : T_1 \rightarrow T_2$ we can define an ordering relation R_1 on the domain of the function as follows: $R_1(x, y) \iff R_2(f(x), f(y))$.

```
-- Map f r : order on t1 induced by mapping t1-elements to
--           t2-elements ordered according to r
Map  :: (t1 → t2) → Order t2 → Order t1
```

3.3 Recursively defined orders

Consider a function that maps order denotations to order denotations on the same type. We introduce a denotation for the order that is the *fixed point* of the function when interpreted as operating on the denoted orders. It can be shown that all order denotations with free variables denote *order-mapping* functions, which constitute the morphisms in the category **TPreorder** of total preorders, which in turn admits (least) fixed points as inverse limits (Henglein 2008).

```
-- Fix rf : fixed-point of order constructor rf
Fix      :: (Order t → Order t) → Order t
```

The Haskell type of `Fix` allows it to be applied to arbitrary *computational* functions mapping order denotations. We shall restrict `Fix` to lambda-abstractions. In other words, `Fix` is intended to be used as μ -notation only.

3.4 Inverse and refinement

Useful additional constructions on orders are: the inverse of an order; and order refinement, where a given order is refined by ordering its equivalence classes according to a second ordering on the same type.

```
-- Inv r : Inverse order of r
--         ((x,y) in Inv r iff (y,x) in r)
Inv      :: Order t → Order t
-- Refine r r' : Refine order r by ordering
--             r-equivalent elements according to r'
Refine   :: Order t → Order t → Order t
```

Other useful denotations could be added, notably for the trivial and empty ordering relation on each type. The trivial ordering relates all elements of a type to each other.

3.5 Bag and set orders

Finite bags (multisets) can be defined as equivalence classes of lists: two lists denote the same bag if one is a permutation of the other. Each list thus becomes a concrete *representative* of a bag: the list $[5, 4, 8]$, understood as a bag representative and for that purpose sometimes written $\langle 5, 4, 8 \rangle$, is equivalent to $[4, 8, 5]$. Similarly, finite sets can be defined by a different equivalence relation: two lists denote the same set if each element in one occurs in the other.

These definitions presuppose a given notion of equality—an equivalence relation, really—on the *elements* of lists. And if the elements are ordered, we can actually define an *ordering relation* on lists understood as bag or set representations:

```
-- Bag r : order on lists induced by r-sorting elements
--         and then lexicographically ordering the
--         resulting lists, also according to r
Bag     :: Order t → Order [t]
-- Set r : order on lists induced by r-sorting elements,
--         eliminating r-duplicates (r-equivalent elements)
--         and then lexicographically ordering
--         the resulting lists, also according to r
Set     :: Order t → Order [t]
```

What makes `Bag r` an ordering relation on lists “understood as bags” is that s_1 and s_2 are related by it if and only if s'_1 and s'_2 are also related for any permutation s'_1 of s_1 and s'_2 of s_2 . Analogous for `Set r`.

3.6 Definable orders

Using the order constructors introduced, other useful orders and order constructors are definable; e.g., standard ascending and descending orders on bytes; lexicographic order on pairs, but with the second component dominant instead of the first; and sum type elements ordered with right summand first instead of left.

```
-- Standard ascending order on [ 0 .. 255 ]
byte    :: Order Int
byte = Nat 256
-- Standard descending order on [ 0 .. 255 ]
byteDown :: Order Int
byteDown = Inv byte
-- Lexicographic ordering on pairs,
-- but with second component dominant
pair2   :: Order t1 → Order t2 → Order (t1, t2)
pair2 r1 r2 = Map swap (Pair r2 r1)
  where swap :: (t1, t2) → (t2, t1)
        swap (x, y) = (y, x)
-- Ordering of tagged values,
-- but with Right elements first
sum2    :: Order t1 → Order t2 → Order (Either t1 t2)
sum2 r1 r2 = Map flip (Sum r2 r1)
  where flip :: Either t1 t2 → Either t2 t1
        flip (Left x) = Right x
        flip (Right y) = Left y
-- Inverse of Sum-ordering
invSum  :: Order t1 → Order t2 → Order (Either t1 t2)
invSum r1 r2 = Inv (Sum r1 r2)
-- An equivalent definition of InvSum
invSum' :: Order t1 → Order t2 → Order (Either t1 t2)
invSum' r1 r2 = sum2 (Inv r1) (Inv r2)
```

Note that the same order may have multiple denotations.

We have introduced order denotations for lists where the list element order is ignored, but, somewhat curiously, not the standard list ordering, which orders lists lexicographically. The reason for this is that it is definable using the already introduced constructions:

```
-- fromList: unfold-part of isomorphism
--           between [t] and Either () (t, [t])
fromList :: [t] → Either () (t, [t])
fromList [] = Left ()
fromList (x : xs) = Right (x, xs)
-- Lexicographic ordering on lists,
-- with elements ordered according to r
list     :: Order t → Order [t]
list r = Fix (\p → Map fromList (Sum Unit (Pair r p)))
```

Note that `fromList` represents (one half of) the isomorphism between lists $[v]$ and `Either () (v, [v])`. Informally, the function that maps `p` to

`Map fromList (Sum Unit (Pair r p))` extends the lexicographic ordering on lists of size n to lists of length $n + 1$. The fixed point thus denotes lexicographic ordering on lists of arbitrary (finite!) length.

4. Generic discrimination

In this section we shall develop a series of generic functions on order denotations: comparison, distributed sorting and finally discrimination. Doing it in these steps is intended to illustrate how the notion of discriminator arises as a natural abstraction.

4.1 Comparison

Generic sorting can be implemented by first defining a generic *comparison* function and then passing that function as an argument to a comparison-based sorting algorithm.

A comparison function is a function that has the right type, $T \times T \rightarrow \text{Bool}$, and that furthermore is transitive and total. Actually, sorting algorithms parameterized on a comparison function can be applied to arbitrary functions as long as they have the right type, whether or not they are comparison functions. In fact, doing so is not only a mistake that goes undiscovered until run-time, it also leaks information on which sorting algorithm is used in the implementation: Any stable sorting algorithm behaves identically when given a *comparison* function, but not on *all* functions of type $T \times T \rightarrow \text{Bool}$.

If the implicit “trust me”-assertion issued by the programmer coding the alleged comparison function is deemed insufficient, a certificate is required that guarantees that the function passed to the sorting algorithm is a bona-fide comparison function.⁵

An order denotation can serve as such a certificate: once type checked (which serves as certificate checking) bona-fide comparison functions can be defined generically from order denotations.

The generic comparison function is defined (co-)inductively, by case analysis on order denotations:

```
-- Generic definition of comparison function
-- (characteristic function on orders)
lte :: Order t => t -> t -> Bool

lte Char x y      = x <= y
lte (Nat n) x y   = x <= y
lte Unit x y      = True
lte (Sum r1 r2) (Left x) (Left y) = lte r1 x y
lte (Sum r1 r2) (Left x) (Right y) = True
lte (Sum r1 r2) (Right x) (Left y) = False
lte (Sum r1 r2) (Right x) (Right y) = lte r2 x y
lte (Pair r1 r2) (x1, x2) (y1, y2) =
  lte r1 x1 y1 &&
  if lte r1 y1 x1
  then lte r2 x2 y2
  else True
lte (Fix rf) x y = lte (rf (Fix rf)) x y
lte (Map f r) x y = lte r (f x) (f y)
lte (Bag r) xs ys =
  lte (list r) (sort r xs) (sort r ys)
lte (Set r) xs ys =
  lte (list r) (usort r xs) (usort r ys)
lte (Refine r r') x y =
  lte r x y &&
  if lte r y x then lte r' x y else True
lte (Inv r) x y = lte r y x
```

In the definition of the comparison function for bag- and set-ordered lists we make use of sorting and unique-sorting functions. A unique-sorting function for ordering relation R returns only one element from each class of R -equivalent elements in the input. Sorting and unique-sorting functions are easily defined mutually recursively with the generic comparison function by employing a standard comparison-based sorting algorithm `sortBy`.

⁵ Alternatively, in principle the sorting algorithm could monitor the alleged comparison function and exit with an error once it has observed that it is not transitive or total. Due to ordinarily applicable performance requirements on sorting this is normally not a viable solution, however.

```
-- Three-valued comparison function
-- from comparison function lte
comp r x y = if lte r x y
             then if lte (Inv r) x y then EQ else LT
             else GT

-- Comparison-based sorting
sort :: Order t => [t] -> [t]
sort r xs = sortBy (comp r) xs

-- Comparison-based sorting,
-- with elimination of equivalent values
usort :: Order t => [t] -> [t]
usort r xs =
  map head (groupBy (lte (Inv r)) (sort r xs))
```

Note that, here, it is the *comparison function* `lte` that is properly *generically* defined, not the sorting function `sort`, which is *parametric* in its argument.

It is an easy exercise to develop `comp` into a combinator library for constructing comparison functions by partially evaluating it on order denotations. See Section 7 where this is done for discriminators.

We believe that, in practice, almost all comparison functions passed to a sorting function such as `sortBy` can be understood as being built from these combinators. This means such comparison functions can also be specified by order denotations and thus by syntactic certificates that guarantee a bona-fide comparison function.

4.2 Sorting

Intuitively, a sorting function for order (S, R) is a function $f : S^* \rightarrow S^*$ that permutes its input such that the resulting output is R -ordered. This is not a good basis for a *generic* definition of sorting, however. Assume we are given sorting functions `sort1` and `sort2` for `r1 :: Order T1` and `r2 :: Order T2`, and we would like to define a sorting function for `Pair r1 r2 :: Order (T1, T2)` in terms of `sort1` and `sort2`. Given a list of pairs we can only sort the first components in isolation using `sort1` or the second components by themselves using `sort2`. This is practically useless, however: we need `sort1` and `sort2` to sort *pairs* of values (k_1, k_2) , but according to the order on k_1 , respectively k_2 . In other words, a sorting function must be able to sort *keys* not only by themselves, but *with associated data values*,

DEFINITION 4.1. [Sorting function] A (*key*) *sorting function* for order (S, R) is a parametric polymorphic function $\text{sort} : \forall \alpha. (S \times \alpha)^* \rightarrow (S \times \alpha)^*$ such that *sort* permutes its input into a list where the first components (keys) are R -ordered.

It is *stable* if R -equivalent elements in the output occur in the same relative order as in the input. \square

The polymorphic type for the value components associated with keys captures that the sorting function works with values of any type and treats them parametrically.

Now we can sort elements whose keys are pairs by first sorting according to the *second components* of the keys and then sorting the result according to the first components, assuming the final sorting step is stable. This works generically if each ordered primitive type, including user-defined abstract types, comes equipped with a stable sorting function. In particular we may use linear-time *distributive* sorting algorithms for the primitive types. This results in the following generalization of distributive sorting to arbitrary denotable orders over first-order types

```
-- Generic definition of
-- stable sorting function for order denotation
dsort :: Order k => [(k, v)] -> [(k, v)]

dsort r [] = []
dsort r (xs @ [(k, v)]) = xs
```

```

dsort Char xs      = bsortChar xs
dsort (Nat n) xs  = bsortNat n xs
dsort Unit xs     = xs
dsort (Sum r1 r2) xs =
  [ x1 | (_, x1) ← dsort r1
    [ (k1, x1) | x1 @ (Left k1, _) ← xs ] ]
++ [ x2 | (_, x2) ← dsort r2
    [ (k2, x2) | x2 @ (Right k2, _) ← xs ] ]
dsort (Pair r1 r2) xs =
  [ x | (_, x) ← dsort r1
    [ (k1, x) | x @ ((k1, k2), v) ← xs' ] ]
  where xs' = [ x | (_, x) ← dsort r2
    [ (k2, x) | x @ ((k1, k2), v) ← xs ] ]
dsort (Fix rf) xs = dsort (rf (Fix rf)) xs
dsort (Map f r) xs =
  [ x | (_, x) ← dsort r [ (f k, x) | x @ (k, v) ← xs ] ]
dsort (Bag r) xs = dsort (Map (sort r) (list r)) xs
dsort (Set r) xs = dsort (Map (usort r) (list r)) xs
dsort (Refine r r') xs = dsort r (dsort r' xs)
dsort (Inv r) xs = reverse (dsort r xs)
bsortChar xs =
  [ (k, v) | (k, vs) ← blocks, v ← reverse vs ]
  where blocks = assocs (accumArray (\vs v → v : vs)
    [] ('\000', '\255') xs)
bsortNat n xs =
  [ (k, v) | (k, vs) ← blocks, v ← reverse vs ]
  where blocks = assocs (accumArray (\vs v → v : vs)
    [] (0, n-1) xs)
-- Sorting on keys only
sort :: Order t → [t] → [t]
sort r xs = [ k | (k, _) ← dsort r [ (k, ()) | k ← xs ] ]
-- Unique sorting
usort :: Order t → [t] → [t]
usort r xs = map head (groupBy (lte (Inv r)) (sort r xs))
-- Comparison function: Parametrically defined from dsort
lte :: Order t → t → t → Bool
lte r x y = res
  where (_, res) = head (dsort r [(x, True), (y, False)])

```

`bsortChar` and `bsortNat` are sorting functions based on bucket sorting. These are distributive sorting algorithms not subject to the information-theoretic lower bound for comparison-based sorting algorithms: they execute in linear time on fixed-width RAMs. Our generic definition of `dsort` bootstraps these basic distributive sorting algorithms to arbitrary (denotable) orders.

The particular Haskell code given for `bsortChar` and `bsortNat` is not interesting. There are more efficiently engineered implementations, which are easiest to formulate in C-style notation since they involve low-level imperative data structures outside the scope of type systems of current strongly typed programming languages. Careful engineering of distributed sorting, especially of its space consumption, is an important, but separate challenge. Attempting so here would obscure the simple generic structure of sorting and discrimination, however.

4.3 Discrimination

Observe the repetitive pattern in the `Pair`-, `Sum`- and `Map`-clauses of `dsort`, where a key is first extracted from each input element, the resulting key-element pairs are sorted and the previously extracted key-components are subsequently discarded. Keys play two separate roles in sorting: they guide the sorting, but get discarded as soon as they have done so; and they are part of the data that are parametrically rearranged during sorting. If we are not interested in the keys in the output we can discard them instantaneously during sorting. For example, the code for the `Sum`-clause then becomes

```

dsort (Sum r1 r2) xs =
  dsort r1 [ (k1, v1) | (Left k1, v1) ← xs ]
++ dsort r2 [ (k2, v2) | (Right k2, v2) ← xs ]

```

and the type of the function is changed from $(T \times \alpha)^* \rightarrow (T \times \alpha)^*$ to $(T \times \alpha)^* \rightarrow \alpha^*$: key-value pairs are input, but only the value-components are returned, sorted according to their in the output and return the keys in the output: Replace each input element (k, v) by $(k, (k, v))$ prior to applying the function, which then simply throws the keys corresponding to the first occurrence of k away, but returns the keys corresponding to the second occurrence of k .

In the generic `Pair` clause of `dsort` above lexicographic sorting has been implemented by right-to-left sorting, as embodied in least-significant-digit (LSD) first radix sorting. A known disadvantage is that LSD always inspects the second component of a key, which may be large, even when its first component is unique within the input to be sorted. To enable left-to-right lexicographic sorting, corresponding to most-significant-digit (MSD) first radix sorting, it is useful to return the result of sorting not as a single list, but as a list of lists, with each element list, termed a *group*, explicitly representing key-equivalent elements.

This finally changes the type from $(T \times \alpha)^* \rightarrow \alpha^*$ to $(T \times \alpha)^* \rightarrow \alpha^{**}$. We call the resulting function a (*sorting*) *discriminator*: It sorts its input according to the key components and returns the result as groups of key-equivalent elements, but without the keys themselves. They are stable as long as the discriminators for primitive orderings are so.

```

-- Generic definition of stable sorting discriminators
disc :: Order k → [(k, v)] → [[v]]
disc r [] = []
disc r [(k, v)] = [[v]]
disc Char xs = discChar xs
disc (Nat n) xs = discNat n xs
disc Unit xs = [[ v | (_, v) ← xs ]]
disc (Sum r1 r2) xs =
  disc r1 [ (k1, v1) | (Left k1, v1) ← xs ]
++ disc r2 [ (k2, v2) | (Right k2, v2) ← xs ]
disc (Pair r1 r2) xs =
  [ vs | ys ← disc r1 [ (k1,(k2,v)) | ((k1,k2),v) ← xs ],
    vs ← disc r2 ys ]
disc (Fix rf) xs = disc (rf (Fix rf)) xs
disc (Map f r) xs = disc r [(f k,v) | (k,v) ← xs]
disc (Bag r) xs = disc (Map (sort r) (list r)) xs
disc (Set r) xs = disc (Map (usort r) (list r)) xs
disc (Refine r1 r2) xs =
  [zs | ys ← disc r1 [ (k,(k,v)) | (k,v) ← xs ],
    zs ← disc r2 ys ]
disc (Inv r) xs = reverse (disc r xs)
discChar xs = [ reverse vs | vs ← vss, not (null vs) ]
  where vss = elems (accumArray (\vs v → v : vs)
    [] ('\000', '\255') xs)
discNat n xs = [ reverse vs | vs ← vss, not (null vs) ]
  where vss = elems (accumArray (\vs v → v : vs)
    [] (0, n-1) xs)

```

Given a sorting discriminator, both (sorting) partitioning, sorting and unique-sorting functions are easily defined.

```

-- Sorting partitioner
-- from sorting discriminator
part :: (Order t) → [t] → [[t]]
part r xs = disc r [ (x, x) | x ← xs ]
-- Sorting function
-- from sorting partitioner
sort :: Order t → [t] → [t]
sort r xs = [ y | ys ← part r xs, y ← ys ]
-- Unique-sorting function
-- from sorting partitioner
usort :: Order t → [t] → [t]
usort r xs = [ head ys | ys ← part r xs ]

```

Likewise, comparison functions can be defined parametrically from discriminators: a comparison function is essentially just a

discriminator or sorting function applied to 2 elements instead of n elements.

```
-- Boolean comparison function
-- from sorting discriminator
lte :: Order t -> t -> t -> Bool
lte r x y = head (concat (disc r [(x, True), (y, False)]))
```

Note that we have turned comparison-based sorting on its head here: in comparison-based sorting it is the comparison function that is defined generically, and the sorting function is defined parametrically from it. Here it is the sorting function—actually, the discriminator—that is defined generically, and the comparison function is defined parametrically from the discriminator!

5. Complexity

Assume each element of a primitive type has a *size* function denoted $| \cdot |$ that maps each member of the type to a natural number. The size function can be extended to elements of pair, sum and recursive types in a natural fashion:

- the size of a pair is the sum of the sizes the elements plus one;
- the size of a sum-tagged value is one plus the size of the underlying untagged value;
- the size of a value as an element of a recursive type is one plus its size as the member of the unfolded type;
- the size of a pointer is one, no matter what data it points to.

The details of which constants to use above are not important. What is critical, however, is that the size function for pairs adds the size of each component separately. This means that the size function measures the storage requirements of *unboxed* and, more generally, *unshared* data asymptotically correctly, but *not* of *shared* data: A directed acyclic graph (dag) with n elements may represent a tree of size $\Theta(2^n)$. The size function will consequently yield $\Theta(2^n)$ even though the dag can be stored in space $O(n)$. This is why the top-down discrimination embodied in our generic discriminator gives asymptotically optimal performance only for *unshared* data. Dealing with sharing requires bottom-up multiset discrimination (Paige 1991; Henglein 2003).

Our computational model is a pointer machine with basic operations operating on constant-sized data. In particular, operations on pairs (construction, projection), tagged values (tagging, pattern matching on primitive tags) and iso-recursive types (folding, unfolding) each take constant time if the arguments have a boxed (pointer) representation.

We say discriminator $\text{disc } r$ executes in linear time if, whenever applied to $[(k_1, v_1), \dots, (k_i, v_i), \dots, (k_n, v_n)]$ with the v_i represented as pointers, it runs in time $O(n + \sum_{i=1}^n |k_i|)$.

5.1 Nonrecursive orders

The question now is: For which values of r does $\text{disc } r$ execute in linear time? Clearly, the function f must execute in linear time if the discriminator for $\text{Map } f \ r$ is to do so, too. Interestingly this is sufficient to guarantee that all Fix -free denotations yield linear-time discriminators.

PROPOSITION 5.1. *Let r be a Fix -free order denotation. If each primitive order denotation has a linear-time discriminator and each function occurring in r executes in linear time then $\text{disc } r$ executes in linear time, too.*

PROOF By induction on r . The key property is that a linear-time executable function f used in $\text{Map } f \ r$ can only increase the size of its output by a constant factor relative to the size of its input. For Bag and Set denotations we also need the fact that $\text{list } r$ yields a linear-time discriminator. \square

It is important to note that the constant factor in the running time of $\text{disc } r$ generally depends on r .

5.2 Recursive orders

To get a sense of when Fix -denotations yield linear-time discriminators, we shall investigate a number of situations where this does not hold.

The most obvious case is $\text{Fix } \lambda p . p :: \text{Order } t$: The resultant discriminator does not terminate. Intuitively, this is because the discriminator does not work on progressively smaller inputs. Functions in $\text{Map } f \ r$ -denotations may even increase the size of arguments to subsequent discriminator calls.

We shall thus stipulate that order denotations be *contractive*. Informally, a *finite approximation* of an order denotation consists of unrolling all occurrences of Fix any finite number of times and then replacing them with a denotation representing the *empty* ordering relation. We say an order denotation $r :: \text{Order } (T)$ is contractive if, for every finite subset S of values of type T , there is a finite approximation of r denoting an order whose domain includes S .

Consider now the order constructor flipflop

```
-- Flip-flop ordering on lists,
flipflop :: Order t -> Order [t]
flipflop r = Fix (\p -> Map (fromList . reverse)
                        (Sum Unit (Pair r p)))
```

It orders lists lexicographically, but not by the standard index order on elements in the list. It first considers the last element of a list, then the first, then next-to-last, second, next-to-next-to-last, third, etc. flipflop Char yields a quadratic time discriminator. It should be noted that also the comparison function $\text{comp } (\text{flipflop Char})$ requires quadratic time. The reason for this is the repeated application of the reverse function.

Let us look at the body of flipflop in more detail:

```
Map (fromList . reverse) (Sum Unit (Pair r p))
```

By Proposition 5.1 we know that this yields a linear-time discriminator as long as p , viewed as a primitive denotation, has a linear-time discriminator. The recursive order denotation flipflop gives rise to a recursively defined discriminator, and the reason for nonlinearity is that the recursive call operates on parts of the input that have already been processed before the recursive call, notably by the reverse function.

The key idea to ensuring linear-time performance of recursive discriminators is the following amortization argument: Make sure that the input can be (conceptually) split such that the execution of the body of a discriminator minus its recursive calls can be charged to one part of the input, and its recursive call(s) to the *other* part.⁶

In other words, the nonrecursive computation steps are not allowed to “touch” those parts of the input that are passed to the recursive call(s): They may maintain and rearrange pointers to those parts, but must not dereference them. How can we ensure that this is obeyed? We insist that the nonrecursive computation steps be *parametric polymorphic* in the parts passed to the recursive call(s) and stipulate a boxed (pointer) representation for data in parametric position (Henglein and Jørgensen 1994).

We require now that parametric polymorphic functions f that are used in order denotations of the form $\text{Map } f \ r$ be *linear-time* and *affine* in data in parametric position. This means that pointers representing data in polymorphic position may be discarded by f , but they must not be duplicated; and that the output must be produced in time linear in the size of the input data *without* counting those parts that are in polymorphic position.

⁶Charging means that we attribute a constant amount of computation to constant amounts of the original input. If all computation steps can be accounted for in this fashion we have proven linear-time complexity.

Since discrimination for `Bag r` and `Set r` denotations is implemented by mapping the sorting, respectively unique-sorting function for `r` over the argument, `r` must not contain occurrences of a `Fix`-bound order variable.

We now have a recipe for constructing order denotations on recursive types that yield linear-time discriminators:

- Let $U = \mu t.T$ be a contractive recursive type with $f : U \rightarrow T[U/t]$ the unfold-part of the isomorphism between U and $T[U/t]$.⁷
- Find an order denotation `r` of polymorphic type $\forall t. \text{Order } t \rightarrow \text{Order } T$ where all functions “mapped” in `r` are linear-time and affine, and occurrences of the form `Bag r1` and `Set r2` in `r` do not contain `p`.
- Form the recursive order denotation `Fix \p. Map f (r[U] p) :: Order U`

The notation `r[U]`, which is not legal Haskell syntax, denotes the explicit instantiation of `r` at type `U`. Note that the isomorphism `f` has the property that it runs in time $O(|v| - |f(v)|)$ for all values v .

This leads to our main complexity theorem:

THEOREM 5.2. *Let `r` be a denotation such that all occurrences of `Fix` are of the form above, primitive discriminators run in linear time, and all mapped functions are linear-time and affine.*

Then `disc r` executes in time $O(|xs|)$ when applied to xs .

PROOF (Idea) By induction on order denotations with free order denotation variables, each of type $\text{Order}(t)$, with t a type variable. \square

Each recursive type has a standard order as long as each primitive type occurring in it is equipped with one: the product type is ordered by `Pair`, the sum type by `Sum` and a recursive type $U = \mu t.T$ by `Fix \p. Map f R` where `R` is the standard order of `T` under the assumption that `p` is the standard order of `t`, and `f` is the unfold-function of U .

As a corollary of Theorem 5.2 we immediately get that all such standard orders yield a linear-time sorting discriminator.

It can be observed that `disc r` executes in linear time if and only if `comp r`, the pairwise comparison function for `r`, does so. In this sense sorting discriminators are a proper generalization of comparison functions: They execute within the same computational resource bounds, but decide the ordering relation on n arguments at a time instead of just 2.

It should be noted here that classical comparison-based sorting algorithms such as Mergesort only yield quadratic time worst-case bounds where the corresponding discriminators for the same order guarantee linear time. (See Section 9 for more on this.)

6. Applications

We present some applications of discrimination, including its variations as sorting and partitioning functions, which are intended to illustrate the expressive power of order denotations and the asymptotic efficiency of generic discrimination, sorting and partitioning.

6.1 Word occurrences

Consider a text. After tokenization we obtain a list of string-integer pairs, where each pair (w, i) denotes that string w occurs at position i in the input text. We are interested in partitioning the indexes such that each equivalence class represents all the occurrences of

⁷ Contractive means that U is not just a list of μ -binders followed by a type variable.

the same word in the text. This is accomplished by the following function:

```
-- occs : Occurrences of identical words in text
occs :: [(String, Int)] -> [[Int]]
occs = disc (list Char)
```

Each group of indexes returned points to the same word in the original text. The order of the index lists returned reflects the lexicographic (alphabetic) order of the words thus indexed. Since all our discriminators are stable if the primitive discriminators are, the indexes of each word are returned in ascending order as long as the input itself is provided in index-ascending order.

If we wish to find occurrences modulo the case of the letters, so the occurrences of “Dog”, “dog” and “DOG” are put into the same equivalence class we simply change the order denotation correspondingly:

```
-- occsCaseIns : As occs, only case insensitive
occsCaseIns :: [(String, Int)] -> [[Int]]
occsCaseIns = disc (list (Map toUpper Char))
```

We could also use `toLower` instead of `toUpper`, which illustrates that the same order may have multiple denotations.

6.2 Anagram classes

A classical problem treated by Bentley in his programming pearls series (Bentley 1983) is *anagram classes*: Given a list of words from a dictionary find their anagram classes; that is find all words that are permutations of each other and do this for all the words in the dictionary. This is tantamount to treating words as bags of characters, and we thus arrive at the following solution:

```
-- anagram classes
anagrams :: [String] -> [[String]]
anagrams = part (Bag Char)
```

Note that the result is returned such that, within each anagram class the words found are alphabetically sorted.

This is bound to be the shortest solution to Bentley’s problem yet, and it even improves his solution *asymptotically*. (His Unix shell-programming solution uses comparison-based sorting.)

If we want to find anagram classes modulo the case of letters we use a modified order denotation, analogous to the way we have done in the word occurrence problem:

```
-- anagram classes, case insensitive
anagramsCaseIns :: [String] -> [[String]]
anagramsCaseIns = part (Bag (Map toUpper Char))
```

Anagram equivalence is bag equivalence for character lists. If we want to find equivalent bags for lists where the elements themselves are sets (also represented as lists, but intended as set representations), which in turn contain bytes, the corresponding ordering relation can be defined as follows:

```
-- ordering relation on lists (as bags)
-- of lists (as sets) of bytes
bsbOrder :: Order [[Int]]
bsbOrder = Bag (Set byte)
```

Discrimination, partitioning and sorting is then simply defined by applying `disc`, `part` and `sort`, respectively, to `bsbOrder`.

6.3 Lexicographic sorting

Let us assume we want to sort strings—lists of characters. Sorting in ascending alphabetic, descending alphabetic and ascending, but case insensitive order can be solved as follows:

```
-- lexicographic sorting functions
lexUp = sort (list Char)
lexDown = sort (list (Inv Char))
lexUpCaseIns = sort (list (Map toUpper Char))
```

Each of these lexicographic sorting functions operates left-to-right and inspects only the characters in the minimum distinguishing prefix of the input; that is, for each input string the minimum prefix required to distinguish the string from all other input strings. (If a string occurs twice, all characters are inspected.) It has the known weakness (Mehlhorn 1984), however, that there are usually many calls to the Char-discriminator with only few arguments. The Char-discriminator returns its input by traversing an array, the *bucket table*, of some fixed size m , which is independent of the number n of arguments passed to it. So traversal time is $O(n + m)$, which means m dominates for small values of n .

If the output does not need to be alphabetically sorted traversal time can be made independent of the array size by employing *basic multiset discrimination* (Cai and Paige 1995, Section 2.2). This motivated Paige and Tarjan to break lexicographic sorting into two phases: In the first phase they identify equal elements, but do not return them in sorted order; instead they build a trie-like data structure. In the second phase they traverse the nodes in this structure in a single sweep and make sure that the children of each node are eventually listed in sorted order, arriving at a proper trie representation of the lexicographically sorted output (Paige and Tarjan 1987, Section 2). Even though building an intermediate data structure such as a trie may at first appear too expensive to be useful in practice, a similar two-phase approach is taken in what is claimed to be the fastest string sorting algorithm for large data sets (Sinha and Zobel 2003).

Another solution is possible, however, which does not require building a trie for the entire input. Consider the code for discrimination of pairs:

```
disc (Pair r1 r2) xs =
  [ vs | ys ← disc r1 [(k1,(k2,v)) | ((k1,k2),v) ← xs],
    vs ← disc r2 ys ]
```

We can see that `disc r2` is called for each group `ys` output by the first discrimination step. If `r2` is `Char`, the repeated calls of `disc r2` are calls to the bucket sorting based discriminator `discChar`. The problem is that each such call may fill the array serving as the bucket table with only few elements before retrieving them by sequential iteration through the entire array. The idea now is to *combine* all calls to `disc r2` into a *single* call by applying it to the *concatenation* of all the groups `ys`. To be able to distinguish from which original group an element comes, each element is tagged with a unique *group number* before being passed to `disc r2`. The output of that call is concatenated and discriminated on the group number they received. This, quite magically, produces the same groups `vs` as in the code above.

Formally, this can be specified as follows:

```
disc (Pair r1 r2) xs =
  disc (Nat (length yss)) (concat (disc r2 zss))
  where yss = disc r1 [ (k1, (k2, v)) |
                      ((k1, k2), v) ← xs ]
        zss = [ (k2, (i, v)) |
              (i, ys) ← zip [0..] yss, (k2, v) ← ys ]
```

The correctness of this code exploits the fact that `disc` is stable. It works for all order denotations. Restricted to the standard lexicographic ordering on strings it is the key idea in Forward Radixsort (Andersson and Nilsson 1994, 1998). It can also be thought of as a local application of least-significant-digit (LSD) sorting, expressed in terms of discriminators.

Going from processing one group at a time to processing *all* of them in one go is questionable from a practical perspective: it is tantamount to going from depth-first processing of groups to breadth-first processing, which has bad locality and low parallelizability. It brings back some of the disadvantages least-significant-digit first radix sorting has vis a vis most-significant-digit. To wit, when us-

ing basic multiset discrimination (Cai and Paige 1995), which does not incur the penalty of traversal of empty buckets, breadth-first group processing has been observed to have noticeably worse practical performance than depth-first processing (Ambus 2004, Section 2.4).

We conjecture that concatenating not *all* groups `ys` returned by `disc r1` in the defining clause for `disc (Pair r1 r2)`, but *just as many* as is necessary to fill the bucket table to “pay” for its traversal will lead to a good algorithm that retains the advantages of MSD radix sorting without suffering the cost of near-empty bucket table traversals.

6.4 Type isomorphism

The type isomorphism problem with an associative type constructor is the problem of partitioning a set of simple types with an associative type constructor \times (product) and with other, free type constructors such as \rightarrow (function type), 0 (Natural numbers).

Even though sorting is not required for the problem we can specify an equivalence relation by providing a total preorder inducing the desired equivalence on the type terms. (A direct specification of equivalence relations is possible, but omitted here.)

The problem can be solved as follows. We define a data type for type expressions:

```
-- data type for representing type expressions
data TypeExp = TCons String [TypeExp]
              | Prod TypeExp TypeExp
```

Here the `Prod` constructor represents the product type constructor; it is singled out from the other type constructors since it is to be treated as an associative constructor.

In the first phase type expressions are normalized such that products occurring in a type are turned into an n -ary product type constructor applied to a list of types, none of which is such a product type. This corresponds to exploiting the associativity property of \times . We can use the following data type for representing the resulting type expressions:

```
-- type expressions with n-ary product type constructor
data TypeExp2 = TCons2 String [TypeExp2]
               | Prod2 [TypeExp2]
```

The normalization function `trans` can be defined as follows:

```
-- transforming Prod-subtrees into Prod2-lists
trans (Prod t1 t2) = Prod2 (traverse (Prod t1 t2) [])
trans (TCons c ts) = TCons2 c (map trans ts)
traverse (Prod t1 t2) rem = traverse t1 (traverse t2 rem)
traverse (TCons c ts) rem = TCons2 c (map trans ts) : rem
```

Having normalized type expressions, the desired equivalence relation is induced by the following total preorder:

```
prod2 :: Order TypeExp2
prod2 = Fix (\p → Map unTypeExp2
                  (Sum (Pair (list Char) (list p)) (list p)))
        where unTypeExp2 (TCons2 v cts) = Left (v, cts)
              unTypeExp2 (Prod2 cts) = Right cts
```

The function `unTypeExp2` is half of the isomorphism between `TypeExp2` and `Either (String, [TypeExp2]) [TypeExp2]`, which is required due to Haskell’s iso-recursive approach to recursive types. Eliding it temporarily, the order `prod2` can be defined recursively as

```
prod2 = Sum (Pair String (list prod2)) (list prod2)
```

where `String = list Char` is the standard order on strings. Reading `prod2` in terms of the equivalence induced, it says that two type expressions are equivalent if and only if they have the same type constructor and their arguments are pairwise equivalent: it is structural equality on `TypeExp2`.

It is easy to see that normalization executes in linear time on *unshared* type expressions, and by Theorem 5.2 the second phase also operates in linear time. It should be noted that the above is the *entire* code of the solution.

A harder variant of this problem is type isomorphism where the product constructor is associative and commutative. Normalization handles associativity as before, and commutativity can be captured by the following order denotation:

```
prod3 :: Order TypeExp2
prod3 = Fix (\p → Map unTypeExp2
  (Sum (Pair (list Char) (list p)) (Bag p)))
  where unTypeExp2 (TCons2 v cts) = Left (v, cts)
        unTypeExp2 (Prod2 cts) = Right cts
```

The only change to prod2 is the use of Bag p instead of list p. Now part prod3 is a solution to the problem. prod3 does not satisfy the requirements of Theorem 5.2, however, and indeed part prod3 does not run in linear time: it takes exponential time!

It has been shown that this problem can be solved in linear time over tree (unboxed) representations of type expressions (Jha et al. 2008) by applying *bottom-up* multiset discrimination for trees with weak sorting (Paige 1991). For pairs of types this has also been proved separately (Zibin et al. 2003), where basic multiset discrimination techniques due to Cai and Paige (1991, 1995) have been rediscovered.

Bottom-up multiset discrimination can handle *shared* acyclic data in linear time. A generic implementation framework for it remains to be developed, however.

7. Combinator library

Since the generic discriminator is defined by (co-)induction over order denotations, order denotations can easily be eliminated by partial evaluation, which results in a combinator library for discriminators. This can be thought of as an exercise in *polytypic programming* (Jeuring and Jansson 1996; Hinze 2000), extended from type denotations (one per type) to order denotations (many per type).

```
-- Discriminator combinators derived from generic definition
-- NB: For simplicity without shortcut clauses
-- for singleton argument lists
type Disc k v = [(k, v)] → [[v]]

discChar :: Disc Char v
discChar xs = [ vs | vs ← vss, not (null vs) ]
  where vss = elems (accumArray (\vs v → v : vs)
    [] ('\000', '\255') xs)

discNat :: Int → Disc Int v
discNat n xs = [ vs | vs ← vss, not (null vs) ]
  where vss = elems (accumArray (\vs v → v : vs)
    [] (0, n-1) xs)

discUnit :: Disc () v
discUnit xs = [[ v | (_, v) ← xs ]]

discSum :: Disc k1 v → Disc k2 v → Disc (Either k1 k2) v
discSum d1 d2 xs =
  d1 [ (k1, v1) | (Left k1, v1) ← xs ]
  ++ d2 [ (k2, v2) | (Right k2, v2) ← xs ]

discPair :: Disc k1 (k2, v) → Disc k2 v → Disc (k1, k2) v
discPair d1 d2 xs =
  [ vs | ys ← d1 [ (k1, (k2, v)) | ((k1, k2), v) ← xs ],
    vs ← d2 ys ]

discFix :: (Disc k v → Disc k v) → Disc k v
discFix df = df (discFix df)

discMap :: (k1 → k2) → Disc k2 v → Disc k1 v
discMap f d xs = d [ (f k, v) | (k, v) ← xs ]

discList :: Disc k [(k, v)] → Disc [k] v
discList d =
  discMap fromList (discSum discUnit
    (discPair d (discList d)))
  where fromList :: [t] → Either () (t, [t])
```

```
fromList [] = Left ()
fromList (x : xs) = Right (x, xs)
discBag :: (forall v. Disc k v) → Disc [k] v
discBag d xs = discMap (sort d) (discList d) xs
discSet :: (forall v. Disc k v) → Disc [k] v
discSet d xs = discMap (usort d) (discList d) xs
discRefine :: Disc k (k, v) → Disc k v → Disc k v
discRefine d1 d2 xs =
  [ zs | ys ← d1 [ (k, (k, v)) | (k, v) ← xs ],
    zs ← d2 ys ]
discInv :: Disc k v → Disc k v
discInv d xs = reverse (d xs)

-- Ordered partitioning from discriminator
part :: Disc t t → [t] → [[t]]
part d xs = d [ (x, x) | x ← xs ]
-- Sorting from ordered partitioning
sort :: Disc t t → [t] → [t]
sort d xs = [ y | y ← part d xs, y ← ys ]
-- Unique sorting from ordered partitioning
usort :: Disc t t → [t] → [t]
usort d xs = [ head ys | ys ← part d xs ]
-- Boolean comparison function
-- from sorting discriminator
lte :: Disc t Bool → t → t → Bool
lte d x y = head (concat (d [(x, True), (y, False)]))
```

The advantage of the discriminator combinator library vis a vis the generic discriminator is that it does away with denotations altogether and lets programmers compose discriminators combinatorially. Also, this incurs no run-time overhead for denotation processing.

The disadvantage is that reasoning about orders and exploiting properties of order denotations is no longer realistically possible. As we have seen, different denotations may denote the same order. For practical reasons one denotation may be preferable over another; e.g., invSum may be preferable to invSum' (see Section 3.6) since it may yield a slightly better performing discriminator. With explicit denotations it is possible for a user to define a function normalize: Order t -> Order t that transforms denotations into an equivalent, for that user presumably better denotation before passing it as an argument to the generic discriminator. Having explicit order denotations thus enables *denotation optimization*, analogous to query optimization, by way of reasoning with and computationally manipulating order denotations prior to instantiating the generic discriminator.

8. Discussion

8.1 Complexity of sorting

The (time) complexity of sorting seems to be subject to some degree of confusion, possibly because different *models of computation* (fixed-word width RAM, RAMs with variable word-width and various word-level operations, cell-probe model, pointer model(s), etc.) and different models of what is *counted* (only number of comparisons in terms of number of elements in input, number of all operations in terms of number of elements, time complexity in terms of size of input) are used, but in each case with the same familiar looking meta-variables (n) and (asymptotic) formulae ($O(n \log n)$).

The quest for *fast integer sorting* in the last 15 years (see Fredman and Willard (1993); Andersson et al. (1998); Han and Thorup (2002) for hallmark results) has sought to perform sorting as (asymptotically) fast as possible as a function of the *number of elements* in the input on RAMs with *variable* word size and *word-level parallelism*.

Our model of computation is a random-access machine with *fixed* word width, say 32 or 64 bits, corresponding to a conventional

Table 1. Comparison-based sorting algorithms for complex data

Sort	Time complexity
Quicksort (Hoare 1961)	$\Theta(N^2)$
Mergesort (Knuth 1998, Sec. 5.2.4)	$\Theta(N^2)$
Heapsort (Williams 1964)	$\Theta(N^2)$
Selection sort (Knuth 1998, Sec. 5.2.3)	$\Theta(N^3)$
Insertion sort (Knuth 1998, Sec. 5.2.1)	$\Theta(N^2)$
Bubble sort (Knuth 1998, Sec. 5.2.2)	$\Theta(N^2)$
Bitonic sort (Batcher 1968)	$\Theta(N \log^2 N)$
Shell sort (Shell 1959)	$\Theta(N \log^2 N)$
Odd-even mergesort (Batcher 1968)	$\Theta(N \log^2 N)$
AKS sorting network (Ajtai et al. 1983)	$\Theta(N \log N)$

sequential computer. (Indeed we only require pointer operations – the random access is not required for our complexity results to hold.) In this setting the only meaningful measure of the input is its *size* in terms of total number of words (or bits) occupied, not the number of elements. If each possible element in an input has constant size, say 32 bits, then input size translates into number of elements, of course. But we want sorting to also work efficiently on inputs with *variable-sized* elements.

An apparently not widely known fact about comparison-based sorting algorithms—I have not seen it stated before—is that they do *not* generally run in time $O(N \log N)$, where N is the size of the input, for inputs with *variable-sized* elements on fixed-width RAMs. It is known that they require $\Theta(n \log n)$ applications of the comparison test, but n is the *number* of elements in the input, not its size, N .

THEOREM 8.1. *Let (A, \leq) be a total preorder and assume that testing whether $v \leq w$ has time complexity $\Theta(|v| + |w|)$. Then comparison-based sorting algorithms have the time complexities given in Table 1.*

PROOF (Proof sketch) The lower bounds for the data-sensitive algorithms (Quicksort, . . . , Bubble sort) follow from analyzing the situation where the input consists of one element of size $\Theta(n)$, with n remaining inputs of size $O(1)$. The upper bounds follow from analyzing how often each element can be an argument in a comparison operation.

Lower and upper bounds for the data-insensitive algorithms (sorting networks) follow from information on their depths and sizes as sorting networks; in particular, the depth provides an upper bound on how many times any given input element is used in a comparison by the algorithm. \square

Note that Mergesort and Heapsort run in *quadratic time* since they run the risk of repeatedly, up to $\Theta(n)$ times, using the same large input element in comparisons, whereas the design of efficient data-insensitive sorting algorithms prevents this. In one important case they run in time $\Theta(N \log N)$, however: lexicographic ordering. In this case the comparison function does need not to investigate all bits in its arguments, only their minimal discriminating prefix.

8.2 Use of explicit fixed point operator

We have used an explicit fixed point operator for recursively defined orders instead of using infinitary terms denoted by recursion equations, as advocated by Hinze (Hinze 2000). So instead of writing

```
list r = Fix \p → Map fromList (Sum Unit (Pair r p))
```

we could define recursively

```
list r = Map fromList (Sum Unit (Pair r (list r)))
```

The use of the explicit fixed-point operator instead of implicit recursion plays a number of roles here. In general, it allows distinguishing between a denotation before and after its unrolling. Of particular importance is that type variables and order denotation variables are interpreted differently from their unrolling in the complexity-theoretic analysis: The parts of an input that “correspond” to a type variable must be treated parametrically—by pointer operations, disallowing their copying—to ensure linear-time performance of the resulting discriminator code.

Finally, even though not pursued here since we only deal with finite unshared data and thus no cyclic data structures, there may be multiple fixed points. Equivalence relations form a complete lattice that admits both greatest and least fixed points of monotonic functions. Extending `list r` as an equivalence denotation to cyclic finite lists we can get distinct fixed points: One where nonidentical, but isomorphic graphs with cycles are considered equivalent (greatest fixed point) and another where they are not (least fixed point).

8.3 Associative reshuffling

The code for discrimination of products contains what looks to be a reshuffling of the input: $((k_1, k_2), v)$ is transformed into $(k_1, (k_2, v))$ before being passed to the first subdiscriminator.

```
disc (Pair r1 r2) xs =
  [ vs | ys ← disc r1 [ (k1, (k2, v)) | ((k1, k2), v) ← xs ],
    vs ← disc r2 ys ]
```

This seems wasteful at first sight. It is an important and in essence unavoidable step, however. It is tantamount to the algorithm moving to the left child of each key pair node and retaining the necessary continuation information. To get a sense of this, let us consider reshuffling in the context of nested products. Consider, for example, `Pair (Pair (Pair r1 r2) r3) r4`, with `r1`, `r2`, `r3`, `r4` being primitive order denotations such as `Char`. The effect of discrimination is that each input $((((k_1, k_2), k_3), k_4), v)$ is initially transformed into $(k_1, (k_2, (k_3, (k_4, v))))$ and then the four primitive discriminators, corresponding to k_1, k_2, k_3, k_4 , are applied in order: The reshuffling ensures that the inputs are lined up in the right order for this.

We may be tempted to perform the reshuffling step lazily, by calling an adapted version `discL` of the discriminator:

```
disc (Pair r1 r2) xs =
  [ vs | ys ← discL r1 xs,
    vs ← disc r2 ys ]
```

But how to define `discL` then? In particular, what to do when *its* argument is, in turn, a product denotation? Introduce `discLL`? At this point we may be tempted to provide an *access* or *extractor* function as an extra argument to a discriminator, as has been done by Ambus (2004). This leads to the following definition of `disc2` for product orders:

```
disc2 (Pair r1 r2) f xs =
  [ vs | ys ← disc2 r1 (fst . f) xs,
    vs ← disc2 r2 (snd . f) ys ]
```

Note that `disc2` takes an access function as an additional argument. The result of `disc2` includes the keys passed to it, and thus the two calls of `disc2` select the first, respectively second component of the key pairs in the input. Since `disc2` is passed an access function `f` to start with the selector functions `fst` and `snd` must be composed with `f`.

In the end this can be extended to a generic definition of `disc2`, which actually sorts its input. It has one critical disadvantage, however: It has, ultimately even asymptotically, inferior performance! The reason for this is that each access to a part of the

input is by navigating to that part from a root node in the original input. The cost of this is thus proportional to the path length from the root to that part. Consider an input element of the form $((\dots((k_1, k_2), k_3), \dots), k_n), v)$, with k_1, \dots, k_n primitive keys. Accessing all n primitive keys by separate accesses, each from the root (the whole value), requires a total of $\Theta(n^2)$ steps!

It is possible to delay or recode the reshuffling step, but it cannot really be avoided.

9. Conclusions

Multiset discrimination has previously been introduced and developed as an algorithmic tool set for efficiently partitioning and preprocessing data according to certain equivalence relations on strings and trees (Paige and Tarjan 1987; Paige 1994; Cai and Paige 1995; Paige and Yang 1997).

We have shown how to analyze multiset discrimination into its functional core components, identifying the notion of discriminator as the core abstraction, and how to compose them generically for a rich class of orders and equivalence relations. In particular, we show that discriminators cannot only be used to partition data, but also to sort them in linear time.

An important aspect of discriminators is that they preserve abstraction: They provide observation of the order (or equivalence relation), but nothing else. This is important when defining an ordered abstract type that should retain as much implementation freedom as possible while providing efficient access to its ordering relation. Discriminators do this, in contrast to *ranking functions*, which map elements to a standard total order such as the integers. The interesting point is that discriminators are principally superior to comparison functions and equality tests: they preserve abstraction, but provide asymptotically improved performance; and to ranking and hash functions: they match their algorithmic performance, but without compromising data abstraction.

We show that type isomorphism for simple types with an associative type constructor can be solved by specifying structural type equality after transforming the input. Type isomorphism with an associative-commutative type constructor (Zibin et al. 2003; Jha et al. 2008) can be expressed, but requires the more complicated bottom-up multiset discrimination to achieve linear-time performance, which is not pursued here. The lexicographic string sorting algorithm of Paige and Tarjan (1987, Section 2) is improved by demonstrating that no clever constant-time array initialization or two-phase processing with explicit trie construction is necessary.

Our work shows that linear-time distributive sorting algorithms, usually restricted to finite domains (bucket sorting) or sequences over such domains (radix sorting) can be bootstrapped to a rich class of orders on arbitrary first-order types, using a single, straightforward generic definition of discrimination. This circumvents the information-theoretic bottleneck of comparison-based sorting algorithms. For types such as ML references that wish to make only an equivalence relation available (setoids) without an ordering relation the argument for discriminators is even more compelling: Discriminators can partition n elements in time $O(n)$. Using a constant-time equality test as only operation to access the equivalence, however, this requires $\Omega(n^2)$ time.

A generic software framework for bulk data processing, such as partitioning and sorting, built on discriminators avoids the algorithmic bottleneck of building it on binary equality/equivalence tests and comparison functions and simultaneously retains their advantages: discriminators are purely functional and disclose only the equivalence relation, respectively order, in question. We conclude that both standard orders and equality relations on types should be exposed as discriminators, not only as binary comparison and equality test functions, as is commonly the case.

It is noteworthy that discriminators *behave* purely functionally and their type can be described as an ML-polymorphic parametric type, but an *efficient* implementation (not demonstrated here, but crucial in practice) of base type discriminators requires highly imperative code that is not typable in an ML-style typing discipline. This may partially explain why discrimination has not previously been discovered in the “natural” course of purely functional programming practice.

9.1 Future work

It is quite easy to see how the definition of generic discrimination can be changed so as to produce in a single pass key-sorted tries instead of just permuted lists of its inputs. This generalizes the trie construction of Paige and Tarjan’s lexicographic sorting algorithm (Paige and Tarjan 1987, Section 2) in two respects: it does so for arbitrary orders, not only for the standard lexicographic order on strings, and it does so in a single pass instead of requiring two. Of particular interest in this connection are Hinze’s generic definitions of operations on generalized tries (Hinze 2000): Discriminators can construct tries in a batch-oriented fashion, and his operations can manipulate them in a one-key-value-pair at a time fashion. There are some differences: Hinze treats nested datatypes, not only regular recursive types, but he has no separate orders or any equivalences on those. In particular, his tries are not key-sorted (the edges out of a node are unsorted). It appears that the treatment of nonnested datatypes can be transferred to discriminators, and the order denotation approach can be transferred to the trie construction operations. We can envisage a generic data structure and algorithm framework where distributive sorting (discrimination) and search structures (tries) supplant comparison-based sorting and comparison-based data structures (search trees), obtaining improved asymptotic time complexities without surrendering data encapsulation. We conjecture that competitive memory utilization and attaining data locality will be serious challenges for the distributive techniques. With the advent of space efficient radix-based sorting (Franceschini et al. 2007), however, we believe that the generic distributive sorting framework presented here can be developed into a framework that has a good chance of outcompeting even highly space efficient in-place comparison-based sorting algorithms in most, if not all, use scenarios of in-memory sorting.

Hinze⁸ has observed that the generic discriminator employs a list monad and that producing a trie is a specific instance of replacing the list monad with another monad, the trie monad. This raises the question of how “general” the functionality of discrimination can be formulated and whether it is possible to characterize discrimination by some sort of *natural* universal property. It also raises the possibility of deforestation-like optimizations: How to avoid building the output lists of a discriminator once we know how they will be destructed in the context of a discriminator application.

Linear-time discrimination for equivalence relations (that is, without sorting) can be extended to shared data for acyclic data structures; discrimination of cyclic data is known to require entirely different algorithmic techniques at the cost of a logarithmic factor (Henglein 2003). Capturing this in a generic programming framework would expand applicability of discrimination to graph isomorphism problems such as deciding bisimilarity, reducing state graphs in model checkers, and the like.

The present functional specification of discrimination has been formulated for clarity, and—most emphatically—not for performance beyond enabling some basic asymptotic reasoning.⁹ Even

⁸Personal communication at IFIP TC.2.8 Working Group meeting, Park City, Utah, June 15-22, 2008.

⁹which would make concrete performance measurements and comparisons not only useless, but outright misguided

though it appears to perform competitively out-of-the-box with good sorting algorithms in terms of time performance, it appears clear that its memory requirements need to—and can—be managed explicitly in a practical implementation. In particular, efficient in-place implementations that do away with the need for dynamic memory management, reduce the memory footprint and improve data locality should provide substantial benefits in comparison to leaving memory management to a general-purpose heap manager.

Join queries and their efficient processing play a central role in database query processing. Discrimination can be used as an alternative to sorting or hashing for their implementation. It may, together with standard data operations such as filtering and selection, provide an interesting generic framework for database programming.

Expressive programming frameworks in languages such as C++, C#, Haskell, Java, OCaml, Python, Scheme, Standard ML, Visual Basic should be developed and evaluated empirically for usability and performance.

Acknowledgements

This paper is dedicated to the memory of Bob Paige. Bob, of course, started it all and got me hooked on multiset discrimination in the first place.

Ralf Hinze alerted me to the possibility of employing GADTs for order denotations by producing an implementation of generic discrimination (for standard orders) in Haskell in no time at all after having seen a single presentation of it at the WG2.8 meeting in 2007. Derek Dreyer has been supportive in the preparation of the final version of the paper, both by discussing the merits of generic discrimination as a fundamental programming abstraction and by debugging my writing.

It has taken me several years and many iterations to generalize—and simultaneously reduce—top-down multiset discrimination into the current, hopefully almost self-evident form. During this time I have had many helpful discussions with a number of people. I would like to thank specifically Thomas Ambus, Martin Elmsan, Hans Leiß and Henning Niss.

The preparation of the submission version was impacted by sickness, and the anonymous referees must be thanked for their patient struggle through it and for providing incisive comments and criticisms. The final version had to be prepared during a planned family vacation on Skopelos, Greece. Upon arrival, my laptop crashed, however. Without my family's patience, the generous help by Kostas and Stratos at Alkistis Hotel, and deep-freezing the laptop to get files off it the paper would not have been completed in time.

References

- M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.
- Thomas Ambus. Multiset discrimination for internal and external data management. Master's thesis, DIKU, University of Copenhagen, July 2004. <http://plan-x.org/projects/msd/msd.pdf>.
- A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–721, 1994.
- Arne Andersson and Stefan Nilsson. Implementing radixsort. *J. Exp. Algorithmics*, 3:7, 1998. ISSN 1084-6654. doi: <http://doi.acm.org/10.1145/297096.297136>.
- Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences (JCSS)*, 57(1):74–93, August 1998.
- K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.
- Jon Bentley. Aha! Algorithms. *Communications of the ACM*, 26(9):623–627, September 1983. Programming Pearls.
- J. Cai and R. Paige. Look ma, no hashing, and no arrays neither. In Jan., editor, *Proc. 18th Annual ACM Symp. on Principles of Programming Languages (POPL)*, Orlando, Florida, pages 143–154, 1991.
- Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science (TCS)*, 145(1-2), July 1995.
- Gianni Franceschini, S. Muthukrishnan, and Mihai Pătraşcu. Radix sorting with no extra space. In *Proc. European Symposium on Algorithms (ESA)*, volume 4698 of *Lecture Notes in Computer Science (LNCS)*, pages 194–205. Springer, 2007. doi: 10.1007/978-3-540-75520-3.
- M.L. Fredman and D.E. Willard. Surpassing the information-theoretic bound with fusion trees. *Journal of Computer and System Sciences (JCSS)*, 47:424–436, 1993.
- Glasgow Haskell. *The Glasgow Haskell Compiler*. <http://www.haskell.org/ghc/>, 2005.
- Yijie Han and Mikkel Thorup. Integer sorting in $o(n\sqrt{\log \log n})$ expected time and linear space. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–144. IEEE Computer Society, 2002.
- Fritz Henglein. Multiset discrimination. Unpublished manuscript. See <http://plan-x.org/msd/multiset-discrimination.pdf>, September 2003.
- Fritz Henglein. A language for total preorders. Unfinished manuscript, March 2008.
- Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *Proc. 21st ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, P.O.Box 64145, Baltimore, MD 21264, Jan. 1994. ACM, ACM Press.
- Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.
- C. A. R. Hoare. Algorithm 63: partition. *Commun. ACM*, 4(7):321, 1961. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/366622.366642>.
- Johan Jeuring and Patrik Jansson. Polytypic programming. In *Advanced Functional Programming*, Lecture Notes in Computer Science, pages 68–114. Springer-Verlag, 1996.
- Sian Jha, Jens Palsberg, Tian Zhao, and Fritz Henglein. Efficient type matching. In Olivier Danvy, Fritz Henglein, Harry Mairson, and Alberto Pettorossi, editors, *Automatic Program Development—A Tribute to Robert Paige*. Springer, 2008. ISBN 978-1-4020-6584-2.
- Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.
- K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume I of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1984.
- R. Paige. Optimal translation of user input in dynamically typed languages. Draft, July 1991.
- Robert Paige. Efficient translation of external input in a dynamically typed language. In *Proc. 13th World Computer Congress*. Elsevier, February 1994.
- Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.
- Robert Paige and Zhe Yang. High level reading and data structure compilation. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, Paris, France, pages 456–469, <http://www.acm.org>, January 1997. ACM, ACM Press.
- D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7), 1959.
- Ranjan Sinha and Justin Zobel. Efficient trie-based sorting of large sets of strings. In Michael Oudshoorn, editor, *Proc. 26th Australasian Computer Science Conference (ACSC)*, Adelaide, Australia, volume 16 of *Conferences in Research and Practice in Information Technology*, 2003.
- J. W. J. Williams. Algorithm 232 - heapsort. *Communications of the ACM*, 7(6):347–348, 1964.
- 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.