Generic Top-down Discrimination for Sorting and Partitioning in Linear Time*

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

December 25th, 2010

Abstract

We introduce the notion of *discrimination* as a generalization of both sorting and partitioning and show that *discriminators* (discrimination functions) can be defined *generically*, by structural recursion on representations of *ordering* and *equivalence relations*.

Discriminators improve the asymptotic performance of generic comparison-based sorting and partitioning, and can be implemented not to expose more information than the underlying ordering, respectively equivalence relation. For a large class of order and equivalence representations, including all standard orders for regular recursive first-order types, the discriminators execute in worst-case linear time.

The generic discriminators can be coded compactly using list comprehensions, with order and equivalence representations specified using Generalized Algebraic Data Types (GADTs). We give some examples of the uses of discriminators, including most-significant-digit lexicographic sorting, type isomorphism with an associative-commutative operator, and database joins. Full source code of discriminators and their applications is included.

We argue that built-in primitive types, notably pointers (references), should come with efficient discriminators, not just equality tests, since they facilitate the construction of discriminators for abstract types that are both highly efficient and representation independent.

*Submitted to JFP. Revised version. 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)* and by the Danish National Advanced Technology Foundation under the grant *3rd generation Enterprise Resource Planning Systems (3gERP)*.

1 Introduction

Sorting is the problem of rearranging an input sequence according to a given total preorder.[1] *Partitioning* is the problem of grouping elements of a sequence into equivalence classes according to a given equivalence relation.

From a programming perspective we are interested in not having to produce hand-written code for each and every total preorder and equivalence relation one may encounter, but to be able to do this *generically*: Specify a total preorder or equivalence relation and automatically generate a sorting, respectively partitioning function, that is both

- *efficient*: it uses few computational resources, in particular it executes fast; and
- *representation independent*: its result is independent of the particular run-time representation of the input data.

Efficiency seems an obviously desirable property, but why should we be concerned with representation independence? The general answer is: Because "data" are not always represented by the "same bits", for either computational convenience or for lack of canonical representation.

Efficiency and representation independence are seemingly at odds with each other. To illustrate this, let us consider the problem of *pointer discrimination*: finding all the duplicates in an input sequence of pointers; that is, partitioning the input according to pointer equality. This is the problem at the heart of persisting ("pickling") pointer data structures onto disk, contracting groups of isomorphic terms with embedded pointers, computing joins on data containing pointers, etc.

Let us try to solve pointer discrimination in ML.[2] Pointers are modeled by *references* in ML, which have allocation, updating, dereferencing and equality testing as the only operations. Representing references as machine addresses at run time, the limited set of operations on ML references guarantees that program execution is semantically deterministic in the presence of nondeterministic memory allocation, and even in the presence of copying garbage collection. In this sense, ML references are *representation independent*: Their operations do not "leak" any observable information about which particular machine addresses are used to represent references at run-time, giving heap allocator and garbage collector free reign to allocate and move references anywhere in memory at any time, without the risk of affecting program semantics.

Having only a binary equality test carries the severe disadvantage, however, that partitioning a list of n references requires $\Theta(n^2)$ equality tests,

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

[2] We use the term ML as a proxy for Standard ML, CaML or any language in the ML family.

which follows from the impossibility of deciding in subquadratic time whether a list of atoms contains a duplicate.

Proposition 1.1 *Let T be a type with at least n distinct elements whose only operation is an equality test. Deciding whether a list of n T-values contains a duplicate requires at least $\binom{n}{2}$ applications of the equality test in the worst case.*

PROOF (By adversary) Assume the problem can be solved using fewer than $\binom{n}{2}$ equality tests. Consider input $[v_1, \ldots, v_n]$ with pairwise distinct input values $v_1, \ldots, v_n$. Then there is a pair v_i, v_j for some i, j with $i \neq j$, for which no equality test is applied. Change the input by replacing v_i with v_j. Now all equality tests performed for the original input give the same result, yet the changed input has a duplicate, whereas the original input does not. □

An alternative to ML references is to abandon all pretenses of guaranteeing representation independence and leaving it in the hands of the developers to achieve whatever level of semantic determinacy is required. This is the solution chosen for object references in Java, which provides a hash function on references.[3] Hashing supports efficient associative access to references. In particular, finding duplicate references can be performed by hashing references into an array and processing the references mapped to the same array bucket one bucket at a time. The price of admitting hashing on references, however, is loss of lightweight implementation of references and loss of representation independence: it complicates garbage collection (e.g. hash values must be stored for copying garbage collectors) and makes execution potentially nondeterministic. Computationally, in the worst case it does not even provide an improvement: All references may get hashed to the same bucket, and unless the hashing function is known to be perfect, pairwise tests are necessary to determine whether they all are equal.

It looks like we have a choice between a rock and a hard place: Either we can have highly abstract references that admit a simple, compact machine address representation and guarantee deterministic semantics, but incur prohibitive complexity of partitioning-style bulk operations (ML references); or we can give up on light-weight references and entrust deterministic program semantics to the hands of the individual developers (Java references).

The problem of multiple run-time representations of the same semantic value is not limited to references. Other examples are abstract types that do not have an unchanging "best" run-time representation, such as sets and bags (multisets). For example, it may be convenient to represent a set by any

[3] We use Java as a proxy for any language that allows interpreting a pointer as a sequence of bits, such as C and C++, or provides a hashing-like mapping of references to integers, such as Java and C#.

list containing its elements, possibly repeatedly. The individual elements in a set may themselves have multiple representations over time or at the same time; e.g., if they are references or are themselves sets. The challenge is how to perform set discrimination efficiently such that the result does not leak information about the particular lists and element representations used to represent the sets in the input.

In this paper we show that execution efficiency and representation independence for *generic* sorting and partitioning can be achieved *simultaneously*. We introduce a bulk operation called *discrimination*, which generalizes partitioning and sorting: It partitions information associated with keys according to a specified equivalence, respectively ordering relation on the keys. For ordering relations, it returns the individual partitions in ascending order.

As Proposition 1.1 and the corresponding well-known combinatorial lower bound of $\Omega(n \log n)$ (Knuth 1998, Section 5.3.1) for comparison-based sorting show, we cannot accomplish efficient generic partitioning and linear-time sorting by using black-box binary comparison functions as *specifications* of equivalence or ordering relations. Instead, we show how to construct efficient discriminators by structural recursion on specifications defined compositionally in an expressive domain-specific language for denoting equivalence and ordering relations.

Informally, generic top-down discrimination for ordering relations can be thought of as filling in the empty square in the diagram below:

Sorting	comparison-based	distributive
Fixed order	Quicksort, Mergesort, etc., with inlined comparisons	Bucketsort, Counting sort, Radixsort
Generic	Comparison-parameterized Quicksort, Mergesort, etc.	

In particular, it extends distributive worst-case linear-time sorting algorithms to all standard orders on all regular recursive first-order types, including tree data structures.

The main benefit of generic discrimination is not for sorting, however, but for partitioning on types that have no natural ordering relation or where the ordering is not necessary: It can reduce quadratic time partitioning based on equality testing to linear time, without leaking more information than pairwise equivalences in the input.

1.1 Contributions

In this paper we develop the notion of *discrimination* as a combination of both partitioning and sorting. Discrimination can also be understood as a generalization of binary equivalence testing and order comparisons from 2 to n arguments.

We claim the following as our contributions:

- An expressive language of *order and equivalence representations* for denoting ordering and equivalence relations, with a domain-theoretic semantics.
- Purely functional *generic* definitions of efficient order and equivalence discriminators.
- Representation independence without asymptotic loss of efficiency: The result of discrimination depends only on the pairwise comparisons between keys, not their particular values.
- A general theorem that shows that the discriminators execute in worst-case linear time on fixed-width RAMs for a large class of order and equivalence representations, including all standard orders and equivalences on regular recursive first-order types.
- A novel value numbering technique for efficient discrimination for bag and set orders and for bag and set equivalences.
- Transparent implementation of generic discrimination in less than 100 lines of Glasgow Haskell, employing list comprehensions and Generalized Algebraic Data Types (GADTs), and with practical performance competitive with the best comparison-based sorting methods in Haskell.
- Applications showing how worst-case linear-time algorithms for non-trivial problems can be derived by applying a generic discriminator to a suitable ordering or equivalence representation; specifically, generalized lexicographic sorting, type isomorphism with associative-commutative operators, and generic equijoins.
- The conclusion that built-in ordered value types and types with equality, specifically reference types, should come equipped with an order, respectively equality discriminator to make their ordering relation, respectively equality, efficiently available.

This article is based on Henglein (2008), though with essentially all aspects reworked, and with the following additional contributions: the domain theoretic model of ordering and equivalence relations; the notion of rank and associated proof principle by structural induction on ranks; the ordinal numbering technique for bag and set orders as well as for bag and set equivalences; the explicit worst-case complexity analysis yielding linear-time discriminators; the definition and semantics of equivalence representations; the definition of generic equivalence discriminator `disc` (not to be confused with the `disc` of Henglein (2008), which, here, is named `sdisc`);

the highly efficient basic equivalence discriminator generator `discNat`; the definition, discussion and proof of representation independence; the application of equivalence discrimination to type AC-isomorphism and database joins; the empirical run-time performance evaluation and comparison with select sorting algorithms; the analysis and dependency of comparison-based sorting on the complexity of comparisons; and some minor other additions and removals.

1.2 Overview

After notational prerequisites (Section 2) we define basic notions: ordering and equivalence relations (Section 3, and discrimination (Section 4).

Focusing first on ordering relations, we show how to construct new ordering relations from old ones (Section 5) and how to represent these constructions as potentially infinite tree data structures (Section 6). We then define order discriminators by structural recursion over order representations (Section 7 and analyze their computational complexity (Section 8).

Switching focus to equivalence relations, we show how to represent the compositional construction of equivalence relations(Section 9), analogous to the development for ordering relations. This provides the basis for generic equivalence discrimination (Section 10). We analyze the representation independence properties of the discriminators (Section 11) before illustrating their use on a number of paradigmatic applications (Section 12). We show that the practical performance of our straightforwardly coded discriminators in Haskell is competitive with sorting (Section 13) and discuss a number of aspects of discrimination (Section 14) before offering conclusions as to what has been accomplished and what remains to be done.

On first reading the reader may want to skip to Sections 6, 7, 12 and 13 to get a sense of discrimination, its applications and performance from a programming point of view.

2 Prerequisites

2.1 Basic mathematical notions

Let $R, Q \subseteq T \times T$ be binary relations over a set T. We often use infix notation: $x\,R\,y$ means $(x, y) \in R$. The *inverse* R^{-1} of R is defined by $x\,R^{-1}\,y$ if and only if $y\,R\,x$. The *restriction* $R|_S$ of R to a set S is defined as $R|_S = \{(x, y) \mid (x, y) \in R \wedge x \in S \wedge y \in S\}$. $R \times Q$ is the pairwise extension of R and Q to pairs: $(x_1, x_2)\,R \times Q\,(y_1, y_2)$ if and only if $x_1\,R\,y_1$ and $x_2\,Q\,y_2$. Similarly, R^* is the pointwise extension of R to lists: $x_1 \ldots x_m\,R^*\,y_1 \ldots y_n$ if and only if $m = n$ and $x_i\,R\,y_i$ for all $i = 1 \ldots n$. We write $\vec{x} \cong \vec{y}$ if $\vec{y}$ is a permutation of the sequence $\vec{x}$.

We assume a universe of discourse U, which contains all objects of interest. A *regular recursive first-order type* is a formal term for denoting a *set* of values. Such types are built from unit (1), product ($\times$), and sum ($+$) constructors; type variables and μ-abstraction for recursive definition; the integers Int. They are inhabited by finite values generated by the grammar

$$v ::= c \mid () \mid \mathit{inl}\, v \mid \mathit{inr}\, w \mid (v, v') \mid \mathit{fold}(v)$$

where $c \in \mathsf{Int}$ is an integer constant. In applications, other primitive types and constants may be added.

Recursive types are interpreted iso-recursively: All values of a recursive type are of the form $\mathit{fold}(v)$, mimicking Haskell's way of defining recursive types by way of `newtype` and `data` declarations. E.g., the list type constructor is defined as $T^* = \mu t.\, 1 + T \times t$, where we define $[] = \mathit{fold}(\mathit{inl}\,())$ and $x :: \vec{x} = \mathit{fold}(\mathit{inr}\,(x, \vec{x})$ and use the notational convention $[x_1, \ldots, x_n] = x_1 :: \ldots :: x_n :: []$.

Type recursion is interpreted *inductively*. In particular, all lists and trees that can occur as *keys* are *finite* in this paper. For emphasis, we note that these types denote sets without any additional structure, such as an element representing nontermination. We allow ourselves to use types also in place of the sets they denote. (Only in Section 8 we treat types as syntactic objects; otherwise they can be thought of as set denotations.)

We use Big-O notation in the following sense: Let f and g be functions from some set S to $\mathbb{R}$. We write $f = O(g)$ if there are constants $a, b \in \mathbb{R}$ such that $f(x) \leq a \cdot g(x) + b$ for all $x \in S$.

We assume basic knowledge of concepts, techniques and results in domain theory, algorithmics and functional programming.

2.2 Haskell notation

To specify concepts and simultaneously provide an implementation for ready experimentation, we use the functional core parts of Haskell (Peyton Jones 2003) as our programming language, extended with Generalized Algebraic Data Types (GADTs), as implemented in Glasgow Haskell (Glasgow Haskell). GADTs provide a convenient *type-safe* framework for shallow embedding of *little languages* (Bentley 1986), which we use for a type-safe coding of ordering and equivalence representation as potentially infinite trees. Hudak et al. (1999) provide a brief and gentle introduction to Haskell, but as we deliberately do not use monads, type classes or any other Haskell-specific language constructs except for GADTs, we believe basic knowledge of functional programming is sufficient for understanding the code we provide.

We are informal about the mapping from Haskell notation to its semantics. As a general convention, we use fixed-width font identifiers for Haskell syntax and write the identifier in italics for what is denoted by it. We use Haskell's built-in types and facilities for defining types, but emphasize that

keys drawn from these types here *are assumed to be belong to the inductive subset* of their larger, coinductive interpretation in Haskell. In particular, only finite-length lists can be keys here.

Haskell's combination of compact syntax, support for functional composition, rich type system, and comparatively efficient implementation constitute what appears to us to presently be the best available uniform framework for supporting the semantic, algorithmic, programming, application and empirical aspects of generic discrimination developed in this paper. It should be emphasized, however, that this paper is about generic discrimination, with Haskell in a support role. The paper is not *about* Haskell in particular, nor is it about developing generic top-down discrimination specifically *for* Haskell. We hope, however, that our work informs future language and library designs, including the Haskell lineage.

2.3 Disclaimer

This paper emphasizes the compositional programming aspects of top-down generic discrimination. It addresses semantic, algorithmic, empirical and application aspects in support of correctness, expressiveness, and computational efficiency, but we avoid detailed descriptions of mathematical concepts and only sketch proofs. A proper *formalization* of the results claimed here in the sense of being worked out in detail and, preferably, in machine-checkable form is not only outside the scope and objective of this paper, but also what we consider a significant challenge left for future work.

3 Ordering and equivalence relations

Before we can introduce discriminators we need to define what exactly we mean by ordering and equivalence relations.

3.1 Ordering relations

Definition 3.1 [Definition set] The *definition set* $\mathrm{def}(R)$ of a binary relation R over S is defined as $\mathrm{def}(R) = \{x \in S \mid (x, x) \in R\}$. □

Definition 3.2 [Ordering relation] A binary relation $R \subseteq S \times S$ is an *ordering relation* over S if for all $x, y, z \in S$:

1. $((x, y) \in R \wedge (y, z) \in R) \Rightarrow (x, z) \in R$ (transitivity), and
2. $((x, x) \in R \vee (y, y) \in R) \Rightarrow ((x, y) \in R \vee (y, x) \in R)$ (conditional comparability).

□

Note that the condition for comparability is disjunctive: Only one of x, y must relate to itself before it relates to every element in S. An alternative is replacing is by a conjunction $((x,x) \in R \wedge (y,y) \in R)$. The present definition is stronger, and we use it since it is noteworthy that the order constructions of Section 5 are closed under this definition.

Not insisting on reflexivity in the definition of ordering relations is important for being able to treat them as pointed directed complete partial orders (dcpos) below.

A word on nomenclature: An ordering relation is not necessarily antisymmetric, so it is a kind of preorder, though not quite, since it is not necessarily reflexive on all of S, only on a subset, the definition set. Analogous to the use of "partial" in partial equivalence relations, we might call it a partial preorder. This would confuse it with "partial order", however, where "partial" is used in the sense of "not total". Note that conditional comparability implies totality on the definition set, and we would end up with something called a partial total preorder, which is not attractive. For this reason we just call our orders "ordering relations". Formally, an *order* is the pair consisting of a set and an ordering relation over that set; analogously for *equivalence*. We informally use "order" and "equivalence" interchangeably with ordering relation and equivalence relation, however.

For ordering relations we use the following notation:

$$\begin{aligned}
x \leq_R y &\Leftrightarrow (x,y) \in R \\
x \geq_R y &\Leftrightarrow y \leq_R x \\
x <_R y &\Leftrightarrow (x,y) \in R \wedge (y,x) \notin R \\
x \equiv_R y &\Leftrightarrow (x,y) \in R \wedge (y,x) \in R \\
x >_R y &\Leftrightarrow y <_R x \\
x \#_R y &\Leftrightarrow (x,y) \notin R \wedge (y,x) \notin R
\end{aligned}$$

Definition 3.3 [Domain of ordering relations over S] The *domain of ordering relations over* S is the pair $(\mathit{Order}(S), \sqsubseteq)$ consisting of the set $\mathit{Order}(S)$ of all ordering relations over S, and the binary relation $\sqsubseteq$ defined by $R_1 \sqsubseteq R_2$ if and only if $x <_{R_1} y \Longrightarrow x <_{R_2} y$ and $x \equiv_{R_1} y \Longrightarrow x \equiv_{R_2} y$ for all $x, y \in S$. □

Proposition 3.4 *$(\mathit{Order}(S), \sqsubseteq)$ is a pointed dcpo.*

PROOF Let $\mathcal{D}$ be a directed set of ordering relations. Then the set-theoretic union $\bigcup \mathcal{D}$ is an ordering relation on S. Furthermore, it is the supremum of $\mathcal{D}$. Observe that the empty set is an ordering relation. It is the least element of $\mathit{Order}(S)$ for any S. □

Note that $\sqsubseteq$ is a finer relation than set-theoretic containment: $R_1 \sqsubseteq R_2 \Longrightarrow R_1 \subseteq R_2$, but not necessarily conversely. For example, $\{(x_1, x_2)\} \subseteq$

$\{(x_1, x_2), (x_2, x_1)\}$, but $\{(x_1, x_2)\} \not\sqsubseteq \{(x_1, x_2), (x_2, x_1)\}$. Intuitively, $\sqsubseteq$ disallows weakening a strict inequality $x <_{R_1} y$ to a nonstrict $x \leq_{R_2} y$. This will turn out to be crucial for ensuring that the lexicographic product order construction in Section 5 is monotonic.

3.2 Equivalence relations

Definition 3.5 [Equivalence relation] A binary relation $E \subseteq S \times S$ is an *equivalence relation* over S if for all $x, y, z \in S$:

1. $((x, y) \in E \wedge (y, z) \in E) \Rightarrow (x, z) \in E$ (transitivity), and
2. $(x, y) \in E \Rightarrow (y, x) \in E$ (symmetry).

□

This is usually called a *partial equivalence relation (PER)*, since reflexivity on S is not required. Since a PER always induces an equivalence relation on its definition set, however, we simply drop the "partial" and call all PERs simply equivalence relations hence.

We write $x \equiv_E y$ if $(x, y) \in E$ and $x \not\equiv_E y$ if $(x, y) \notin E$.

Definition 3.6 [Domain of equivalence relations over S] The *domain of equivalence relations over* S is the pair $(\mathit{Equiv}(S), \subseteq)$ consisting of the set $\mathit{Equiv}(S)$ of all partial equivalence relations on S, together with subset containment $\subseteq$. □

Proposition 3.7 *$(\mathit{Equiv}(S), \subseteq)$ is a pointed dcpo.*

PROOF Let $\mathcal{D}$ be a directed set of equivalence relations Then the set-theoretic union $\bigcup \mathcal{D}$ is an equivalence relation over S. Furthermore, it is the supremum of $\mathcal{D}$. Observe that the empty set is an equivalence relation. It is the least element for $\mathit{Equiv}(S)$ for any S. □

Each ordering relation canonically induces an equivalence relation:

Proposition 3.8 *Let R be an ordering relation. Then $\equiv_R$ is the largest equivalence relation contained in R.*

4 Discrimination

Sorting, partitioning and discrimination functions can be thought of as variations of each other. The output of a *sorting function* permutes input keys according to a given ordering relation. A *partitioning function* groups the input keys according to a given equivalence relation. A *discrimination function (discriminator)* is a combination of both, though with a twist: Its input are key-value pairs, but only the value components are returned in the output.

Definition 4.1 [Values associated with key] Let $\vec{x} = [(k_1, v_1), \ldots, (k_n, v_n)]$. Let R be an ordering or equivalence relation. Then the *values associated with k under R in $\vec{x}$* is the list

$$\mathrm{vals}_R^{\vec{x}}(k) = \texttt{ map snd (filter } (p_R(k))\ \vec{x}\texttt{)}$$

where $p_R(k)(k', v') = (k \equiv_R k')$. □

Note that the values in $\mathrm{vals}_R^{\vec{x}}(k)$ are listed in the same order as they occur in $\vec{x}$.

Definition 4.2 [Discrimination function] A partial function $\mathcal{D} : (S \times U)^* \hookrightarrow U^{**}$ is a *discrimination function for equivalence relation E* if E is an equivalence relation over S, and

(a) $\texttt{concat}\ (\mathcal{D}(\vec{x})) \cong \texttt{map (map snd)}\ \vec{x}$ for all $\vec{x} = [(k_1, v_1), \ldots, (k_n, v_n)]$ where $k_i \in \mathrm{def}(E)$ for all $i = 1 \ldots n$ (permutation property);

(b) if $\mathcal{D}(\vec{x}) = [b_1, \ldots, b_n]$ then $\forall i \in \{1, \ldots, n\}. \exists k \in \texttt{map fst}\ \vec{x}\,.\, b_i \cong \mathrm{vals}_R^{\vec{x}}(k)$ (partition property);

(c) for all binary relations $Q \subseteq U \times U$, if $\vec{x}\,(id \times Q)^*\,\vec{y}$ and both $\mathcal{D}(\vec{x})$ and $\mathcal{D}(\vec{y})$ are defined, then $\mathcal{D}(\vec{x})\,Q^{**}\,\mathcal{D}(\vec{y})$ (parametricity property).

A discrimination function is also called *discriminator*.

We call a discriminator *stable* if it satisfies the partition property with $\cong$ replaced by $=$; that is, if each block in $\mathcal{D}(\vec{x})$ contains the value occurrences in the same positional order as in $\vec{x}$. □

Definition 4.3 [Order discrimination function] A discriminator $\mathcal{D} : (S \times U)^* \hookrightarrow U^{**}$ for E is an *order discrimination function for ordering relation R* if $E = (\equiv_R)$ and the groups of values associated with a key are listed in ascending key-order (sorting property); that is, for all $\vec{x}, k, k', i, j$, if $\mathcal{D}(\vec{x}) = [b_1, \ldots, b_m]$ and $\mathrm{vals}_R^{\vec{x}}(k) = b_i \wedge \mathrm{vals}_R^{\vec{x}}(k') = b_j \wedge k \leq_R k'$ then $\Longrightarrow i \leq j$. An order discrimination function is also called *order discriminator*. □

What a discriminator does is surprisingly complex to define formally, but rather easily described informally: It treats keys as labels of values and groups together all the values with the same label in an input sequence. The labels themselves are not returned. Two keys are treated as the "same label" if they are equivalent under the given equivalence relation. The parametricity property expresses that values are treated as *satellite data*, as in sorting algorithms (Knuth 1998, p. 4) (Cormen et al. 2001, p. 123) (Henglein 2009, p. 555). In particular, values can be passed as pointers that are not dereferenced during discrimination.

A discriminator is stable if it lists the values in each group in the same positional order as they occur in the input. A discriminator is an order discriminator if it lists the groups of values in ascending order of their labels.

Definitions 4.2 and 4.3 fix to various degrees the positional order of the groups in the output and the positional order of the values inside each group. For order discriminators the positional order of groups is fixed by the key ordering relation, but the positional order inside each group may still vary. Requiring stability fixes the positional order inside each group. In particular, for a stable order discriminator the output is completely fixed.

Example 4.4 Let O_{eo} be the ordering relation on integers such that $x\, O_{eo}\, y$ if and only if x is even or y is odd; that is, under O_{eo} all the even numbers are equivalent and they are less than all the odd numbers, which are equivalent to each other. We denote by E_{eo} the equivalence induced by O_{eo}: Two numbers are E_{eo}-equivalent if and only if they are both even or both odd.

Consider

$$\vec{x} = [(5, \texttt{"foo"}), (8, \texttt{"bar"}), (6, \texttt{"baz"}), (7, \texttt{"bar"}), (9, \texttt{"bar"})].$$

A discriminator $\mathcal{D}_1$ for E_{eo} may return

$$\mathcal{D}_1(\vec{x}) = [[\texttt{"foo"}, \texttt{"bar"}, \texttt{"bar"}], [\texttt{"bar"}, \texttt{"baz"}]] :$$

"`foo`" and "`bar`" are each associated with the odd keys in the input, with "`bar`" being so twice; likewise "`baz`" and "`bar`" are associated with the even keys.

Another discriminator $\mathcal{D}_2$ for E_{eo} may return the groups in the opposite order:

$$\mathcal{D}_2(\vec{x}) = [[\texttt{"bar"}, \texttt{"baz"}], [\texttt{"foo"}, \texttt{"bar"}, \texttt{"bar"}]],$$

and yet another discriminator $\mathcal{D}_3$ may return the groups ordered differently internally (compare to $\mathcal{D}_1$):

$$\mathcal{D}_3(\vec{x}) = [[\texttt{"bar"}, \texttt{"foo"}, \texttt{"bar"}], [\texttt{"baz"}, \texttt{"bar"}]].$$

Note that $\mathcal{D}_3$ does not return the values associated with even keys in the same positional order as they occur in the input. Consequently, it is not stable. $\mathcal{D}_1$ and $\mathcal{D}_2$, on the other hand, return the values in the same order.

Let us apply $\mathcal{D}_1$ to another input:

$$\vec{y} = [(5, 767), (8, 212), (6, 33), (7, 212), (9, 33)].$$

By parametricity we can conclude that

$$\mathcal{D}_1(\vec{y}) = [[767, 212, 33], [212, 33]]$$

or

$$\mathcal{D}_1(\vec{y}) = [[767, 33, 212], [212, 33]].$$

To see this, consider

$$Q = \{(\text{"}\texttt{foo}\text{"}, 767), (\text{"}\texttt{bar}\text{"}, 212), (\text{"}\texttt{baz}\text{"}, 33), (\text{"}\texttt{bar}\text{"}, 33)\}.$$

We have $\vec{y}\,(id \times Q)^*\,\vec{x}$, and thus $\mathcal{D}_1(\vec{y})\,Q^{**}\,\mathcal{D}_1(\vec{x})$ by the parametricity property of discriminators. Recall that $\mathcal{D}_1(\vec{x}) = [[\text{"}\texttt{foo}\text{"}, \text{"}\texttt{bar}\text{"}, \text{"}\texttt{bar}\text{"}], [\text{"}\texttt{bar}\text{"}, \text{"}\texttt{baz}\text{"}]]$. Of the 8 possible values that are Q^{**}-related to $\mathcal{D}_1(\vec{x})$, corresponding to a choice of 212 or 33 for each occurrence of "`bar`", only the two candidates above satisfy the partitioning property required of a discriminator.

An order discriminator for O_{eo} must return the groups in accordance with the key order. In particular, the values associated with even-valued keys must be in the first group. Since $\mathcal{D}_2(\vec{x})$ returns the group of values associated with odd keys first, we can conclude that $\mathcal{D}_2$ is not an order discriminator for O_{eo}.

□

5 Order constructions

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.

We provide a number of *order constructions*, which are the basis of an expressive language for specifying such ordering relations. The following are ordering relations:

- The *empty relation* $\emptyset$, over any set S.
- The *trivial relation* $S \times S$, over any set S.
- For nonnegative n, the standard order

  $$[n] = \{(k, l) \mid 0 \leq k \leq l \leq n\}$$

  over any S such that $\{0, \ldots, n\} \subseteq S \subseteq \mathbb{Z}$.

Given $R_1 \in \mathit{Order}(T_1), R_2 \in \mathit{Order}(T_2), f \in T_1 \to T_2$, the following are also ordering relations:

- The *sum order* $R_1 +_L R_2$ over $T_1 + T_2$, defined by

$$x \leq_{R_1 +_L R_2} y \Leftrightarrow \begin{cases} (x = \mathit{inl}\, x_1 \wedge y = \mathit{inr}\, y_2) \vee \\ (x = \mathit{inl}\, x_1 \wedge y = \mathit{inl}\, y_1 \wedge x_1 \leq_{R_1} y_1) \vee \\ (x = \mathit{inr}\, x_2 \wedge y = \mathit{inr}\, y_2 \wedge x_2 \leq_{R_2} y_2) \\ \text{for some } x_1, y_1 \in T_1, x_2, y_2 \in T_2. \end{cases}$$

The subscript in $+_L$ (for "left") indicates that all the left elements are smaller than the right elements. Left elements are ordered according to R_1, and right elements are ordered according to R_2.

- The *lexicographic product order* $R_1 \times_L R_2$ over $T_1 \times T_2$, defined by

$$(x_1, x_2) \leq_{R_1 \times_L R_2} (y_1, y_2) \Leftrightarrow x_1 <_{R_1} y_1 \vee (x_1 \equiv_{R_1} y_1 \wedge x_2 \leq_{R_2} y_2).$$

  The subscript in $\times_L$ (here for "lexicographic") indicates that the first component in a pair is the dominant component: it is compared first, and only if it is equivalent with the first component of the other pair, the respective second components are compared.

- The *preimage* $f^{-1}(R_2)$ *of* R_2 *under* f, over T_1, defined by

$$x \leq_{f^{-1}(R_2)} y \Leftrightarrow f(x) \leq_{R_2} f(y).$$

- The *lexicographic list order* $[R_1]$, over T_1^*, defined by

$$[x_1, \ldots, x_m] \leq_{[R_1]} [y_1, \ldots, y_n] \Leftrightarrow \\ \exists i \leq m+1.\ ((i = m+1) \vee x_i <_{R_1} y_i) \wedge \forall j < i.\ x_j \equiv_{R_1} y_j$$

- The *lexicographic bag order* $\langle R_1 \rangle$, over T_1^*, defined by

$$\vec{x} \leq_{\langle R_1 \rangle} \vec{y} \Leftrightarrow [x'_1, \ldots, x'_m] \leq_{[R_1]} [y'_1, \ldots, y'_n]$$

  where $\vec{x} \cong [x'_1, \ldots, x'_m]$, $\vec{y} \cong [y'_1, \ldots, y'_n]$ such that $x'_1 \leq_{R_1} \ldots \leq_{R_1} x'_m$ and $y'_1 \leq_{R_1} \ldots \leq_{R_1} y'_n$. In words, it is the ordering relation on lists of type T_1 that arises from first sorting the lists in ascending order before comparing them according to their lexicographic list order.

- The *lexicographic set order* $\{R_1\}$, over T_1^*, defined by

$$\vec{x} \leq_{\{R_1\}} \vec{y} \Leftrightarrow [x'_1, \ldots, x'_k] \leq_{[R_1]} [y'_1, \ldots, y'_l]$$

  where $x'_1 <_{R_1} \ldots <_{R_1} x'_k$ and $y'_1 <_{R_1} \ldots <_{R_1} y'_l$ are maximal length proper R_1-chains of elements from $\vec{x}$ and $\vec{y}$, respectively. In words, it is the ordering relation on lists of type T_1 that arises from first unique-sorting lists in ascending order, which removes all $\equiv_{R_1}$-duplicates, before comparing them according to their lexicographic list order.

- The *inverse* R_1^{-1}, over T_1, defined by

$$x \leq_{R_1^{-1}} y \Leftrightarrow x \geq_{R_1} y.$$

Observe that the *Cartesian product relation* $R_1 \times R_2$ over $T_1 \times T_2$, with pointwise ordering does not define an ordering relation. It satisfies transitivity (it is a preorder on its definition set), but not conditional comparability.

Given dcpos D_1, D_2, recall that $[D_1 \to D_2]$ denotes the dcpo of continuous functions from $D_1 \to D_2$, ordered pointwise.

Theorem 5.1 *Let T_1, T_2 be arbitrary sets. Then:*

$$\begin{aligned}
\times_L &\in [Order(T_1) \times Order(T_2) \to Order(T_1 \times T_2)] \\
+_L &\in [Order(T_1) \times Order(T_2) \to Order(T_1 + T_2) \\
.^{-1} &\in (T_1 \to T_2) \to [Order(T_2) \to Order(T_1)] \\
[.] &\in [Order(T_1) \to Order(T_1^*)] \\
\langle . \rangle &\in [Order(T_1) \to Order(T_1^*)] \\
\{.\} &\in [Order(T_1) \to Order(T_1^*)]
\end{aligned}$$

PROOF By inspection. We require $\sqsubseteq$ as the domain relation on ordering relations since $\times_L$ is nonmonotonic in its first argument under set containment $\subseteq$. □

Corollary 5.2 *Let $F \in Order(T) \to Order(T)$ be a function built by composing order constructions in Theorem 5.1, the argument order and given ordering relations ("constants"). Then $F \in [Order(T) \to Order(T)]$ and thus F has a least fixed point $\mu F \in Order(T)$.*

6 Order representations

In this section we show how to turn the order constructions of Section 5 into a domain-specific language of *order representations*. These will eventually serve as arguments to a generic order discriminator.

6.1 Basic order constructors

Definition 6.1 [Order representation] An *order representation over type T* is a value r of type `Order T` constructible by the generalized algebraic data type in Figure 1, where all arguments $f : T_1 \to T_2$ to `MapO` occurring in a value are total functions (that is, $f(x) \neq \bot$ for all $x \in T_1$) and T_1, T_2 are regular recursive first-order types. □

Order representations are not ordering relations themselves, but tree-like data structures *denoting* ordering relations. We allow infinite order representations. As we shall see, such infinite trees allow representation of ordering relations on recursive types.

```
data Order t where
     NatO     :: Int → Order Int
     TrivO    :: Order t
     SumL     :: Order t1 → Order t2 → Order (Either t1 t2)
     ProdL    :: Order t1 → Order t2 → Order (t1, t2)
     MapO     :: (t1 → t2) → Order t2 → Order t1
     ListL    :: Order t → Order [t]
     BagO     :: Order t → Order [t]
     SetO     :: Order t → Order [t]
     Inv      :: Order t → Order t
```

Figure 1: Order representations

An *order expression* is any Haskell expression, which evaluates to an order representation. This gives us 3 levels of interpretation: A Haskell order *expression* evaluates to an order *representation*, which is a data structure that denotes an ordering *relation*. Note that not all Haskell expressions of type `Order T` are order expressions, but henceforth we shall assume that all expressions of type `Order T` that we construct are order expressions.

6.2 Definable orders

Using the order constructors introduced, many useful orders and order constructors are definable.

The standard order on the unit type `()` is its trivial order, which is also its only order:

```
ordUnit :: Order ()
ordUnit = TrivO
```

The standard ascending order on 8-bit and 16-bit nonnegative numbers are defined using the `NatO`-order constructor:[4]

```
ordNat8 :: Order Int
ordNat8 = NatO 255

ordNat16 :: Order Int
ordNat16 = NatO 65535
```

We might want to use

```
ordInt32W  :: Order Int
ordInt32W = MapO tag (SumL (NatO 2147483648) (NatO 2147483647))
   where tag i = if i < 0 then Left (-i) else Right i
```

[4]Somewhat unconventionally, `NatO` n denotes the ascending standard ordering relation on $\{0 \ldots n\}$, not $\{0 \ldots n-1\}$. This reflects the Haskell convention of specifying intervals in the same fashion; e.g. `newArray (0, 65535) []` allocates an array indexed by $[0 \ldots 65535]$. Using the same convention avoids the need for computing the predecessor in our Haskell code in a number of cases.

to denote the standard ordering on 32-bit 2s-complement integers. (Note that $2^{31} = 2147483648$.) This does not work, since 2147483648 is not a 32-bit 2s-complement representable integer, however. (Because `NatO` has type `Int -> Order Int`, where `Int` denotes the 32-bit 2s-complement representable integers, its argument has to be a 32-bit integer.) Since the arguments of `NatO` are used by our basic discriminator as the size of a table to be allocated at run-time, even if 2147483648 were acceptable, large argument values to `NatO` would be unusable in practice. Instead we use the following order representation for the standard order on `Int`:

```
ordInt32 :: Order Int
ordInt32 = MapO (splitW ∘ (+ (-2147483648))) (ProdL ordNat16 ordNat16)

splitW :: Int → (Int, Int)
splitW x = (shiftR x 16 .&. 65535, x .&. 65535)
```

Here we first add -2^{31}, the smallest representable 32-bit 2s complement integer, and then split the resulting 32-bit word into its 16 high-order and low-order bits. The lexicographic ordering on such pairs, interpreted as 16-bit nonnegative integers, then yields the standard ordering on 32-bit 2s-complement integers. As we shall see, `ordInt32` yields an efficient discriminator that only requires a table with $2^{16} = 65536$ elements.

The standard order on Boolean values is denotable by the canonical function mapping `Bool` to its isomorphic sum type:

```
ordBool :: Order Bool
ordBool = MapO bool2sum (SumL ordUnit ordUnit)
  where bool2sum :: Bool → Either () ()
        bool2sum False = Left ()
        bool2sum True = Right ()
```

Analogously, the standard alphabetic orders on 8-bit and 16-bit characters are definable by mapping them to the corresponding orders on natural number segments:

```
ordChar8 :: Order Char
ordChar8 = MapO ord ordNat8

ordChar16 :: Order Char
ordChar16 = MapO ord ordNat16
```

As an illustration of a denotable nonstandard order, here is a definition of `evenOdd`, which denotes the ordering O_{eo} from Example 4.4:

```
evenOdd :: Order Int
evenOdd = MapO (`mod` 2) (NatO 1)
```

The `SumL` order lists left elements first. What about the dual order constructor, where right elements come first? It is definable:

```
sumR :: Order t1 → Order t2 → Order (Either t1 t2)
sumR r1 r2 = Inv (SumL (Inv r1) (Inv r2)
```

An alternative definition is

```
sumR' r1 r2 = MapO flip (SumL r2 r1)
   where flip :: Either t1 t2 → Either t2 t1
         flip (Left x) = Right x
         flip (Right y) = Left y
```

Similarly, the lexicographic product order with dominant right component is definable as

```
pairR :: Order t1 → Order t2 → Order (t1, t2)
pairR r1 r2 = MapO swap (ProdL r2 r1)
    where swap :: (t1, t2) → (t2, t1)
          swap (x, y) = (y, x)
```

The *refinement* of the equivalence classes of one order by another order is definable as follows:

```
refine :: Order t → Order t → Order t
refine r1 r2 = MapO dup (ProdL r1 r2)
  where dup x = (x, x)
```

For example, the nonstandard total order on 16-bit nonnegative integers, where all the even numbers, in ascending order, come first, followed by all the odd numbers, also in ascending order, is denoted by `refine evenOdd ordNat16`.

6.3 Lexicographic list order

For recursively defined data types, order representations generally need to be recursively defined, too. We first consider `ListL`, the lexicographic list order constructor, and show that it is actually definable using the other order constructors. Then we provide a general recipe for defining orders on regular (nonnested) inductive data types.

Consider the type T^* of lists with elements of type T with an element ordering R denoted by order representation r. We want to define a representation of the lexicographic list order $[R]$. We use Haskell's standard list type constructor `[T]`, with the caveat that only T^*, the *finite* lists, are intended, even though Haskell lists may be infinite.

We know that `[t]` is isomorphic to `Either () (t, [t])`, where

```
fromList ::  [t]       →  Either () (t, [t])
fromList     []        =  Left ()
fromList     (x : xs)  =  Right (x, xs)
```

is the "unfold"-direction of the isomorphism. Assume we have a representation of r' of $[R]$ and consider two lists $\vec{x}, \vec{y}$ where $\vec{x} \leq_{[R]} \vec{y}$. Applying `fromList` to them we can see that the respective results are ordered according to `SumL ordUnit (ProdL` r r'`)`. Conversely, if they are ordered like that, then $\vec{x} \leq_{[R]} \vec{y}$.

This shows that we can define $r' =$ `listL` r where

```
listL :: Order t → Order [t]
listL r = MapO fromList (SumL ordUnit (ProdL r (listL r)))
```

As an illustration of applying `listL`, the standard alphabetic order `ordString8` on `String = [Char]`, restricted to 8-bit characters, is denotable by applying `listL` to the standard ordering on characters:

```
ordString8 :: Order String
ordString8 = listL ordChar8
```

6.4 Orders on recursive data types

The general recipe for constructing an order representation over recursive types is by taking the fixed point of an order constructor: Let $p \in [\texttt{Order}(T) \to \texttt{Order}(T)]$ and take its least fixed point $r = p(r)$. By Proposition 5.2 and standard domain-theoretic techniques (Abramsky and Jung 1992, Lemma 2.1.21) this r exists and denotes the least fixed point of the function on ordering relations represented by p.

As an example, consider the type of node-labeled trees

```
data Tree v = Node (v, [Tree v])
```

with unfold-function

```
unNode :: Tree v → (v, [Tree v])
unNode (Node (v, ts)) = (v, ts)
```

The standard order on trees can be defined as

```
tree :: Order t → Order (Tree t)
tree r = Map unNode (ProdL r (ListL (tree r)))
```

It compares the root labels of two trees and, if they are `r`-equivalent, lexicographically compares their children. This amounts to ordering trees by lexicographic ordering on their preorder traversals. Note that `tree r` denotes the least fixed point of

$$\lambda r'.\ \texttt{Map unNode (ProdL r (ListL } r'\texttt{))}.$$

As an example of a nonstandard order on trees, consider the *level-k order* `treeK k` on trees:

```
treeK :: Int → Order t → Order (Tree t)
treeK 0 r = TrivO
treeK k r = Map unNode (ProdL r (ListL (treeK (k-1) r)))
```

It is the same as `tree`, but treats trees as equivalent if they are the same when "cut off" at level k.

Another example of an ordering relation on trees for a given node ordering is

```
treeB :: Order t → Order (Tree t)
treeB r = MapO unNode (ProdL r (BagO (treeB r)))
```

It treats the children of a node as an unordered bag in the sense that any permutation of the children of a tree results in an equivalent tree. Finally,

```
treeS :: Order t → Order (Tree t)
treeS r = MapO unNode (ProdL r (SetO (treeS r)))
```

treats multiple equivalent children of a node as an unordered set: multiple children that turn out to be equivalent, are treated as if they were a single child.

Whether children of a node are treated as lists, bags or sets in this sense is not built into the data type, but can be freely mixed. For example

```
tree1 r = MapO unNode (ProdL r (ListL tree2 r))
tree2 r = MapO unNode (ProdL r (BagO tree3 r))
tree3 r = MapO unNode (ProdL r (SetO tree1 r))
```

interprets nodes at alternating levels as lists, bags and sets, respectively.

6.5 Denotational semantics of order representations

So far we have informally argued that each order representation denotes an ordering relation. In this section we provide the mathematical account of this. Basically, we do this by interpreting each order constructor as the corresponding order construction. Since order representations can be infinite trees, we need to be a bit careful. We can leverage our domain-theoretic framework: We approximate each order representation by cutting it off at level k, show that the interpretations form an ω-chain, and define the interpretation of a order representation as the supremum of its level-k approximations. Even though, domain-theoretically, the development below is entirely standard, we provide an explicit account of it as it forms the basis of the definition of rank, which provides the basis for inductive proofs for structurally recursively defined functions on order representations.[5]

Definition 6.2 [Level-k approximation of order representation] The *level-k approximation* $r|_k$ of order representation r is defined as follows:

$$\begin{aligned}
r|_0 &= \bot \\
(\texttt{NatO}\ m)_{n+1} &= \texttt{NatO}\ m \\
\texttt{TrivO}_{n+1} &= \texttt{TrivO} \\
(\texttt{SumL}\ r_1\ r_2)_{n+1} &= \texttt{SumL}\ r_1|_n\ r_2|_n \\
(\texttt{ProdL}\ r_1\ r_2)_{n+1} &= \texttt{ProdL}\ r_1|_n\ r_2|_n \\
(\texttt{MapO}\ f\ r)_{n+1} &= \texttt{MapO}\ f\ r|_n
\end{aligned}$$

[5] This can be thought of as Scott induction, extended to make statements about termination.

$$
\begin{aligned}
(\texttt{ListL}\ r)_{n+1} &= \texttt{ListL}\ r|_n \\
(\texttt{BagO}\ r)_{n+1} &= \texttt{BagO}\ r|_n \\
(\texttt{SetO}\ r)_{n+1} &= \texttt{SetO}\ r|_n \\
(\texttt{Inv}\ r)_{n+1} &= \texttt{Inv}\ r|_n
\end{aligned}
$$

for all $m, n \geq 0$, where $\perp$ denotes error or nontermination, as in Haskell. □

Note that $r|_n$ is a finite tree of maximum depth n.

Recall the definition of order constructions from Section 5.

Definition 6.3 [Ordering relation denoted by order representation] Let $\mathcal{O}[\![r]\!]$ on finite order representations r be defined inductively as follows:

$$
\begin{aligned}
\mathcal{O}[\![\perp]\!] &= \emptyset \\
\mathcal{O}[\![\texttt{NatO}\ m]\!] &= [m] \\
\mathcal{O}[\![\texttt{TrivO} :: \texttt{Order T}]\!] &= T \times T \\
\mathcal{O}[\![\texttt{SumL}\ r_1\ r_2]\!] &= \mathcal{O}[\![r_1]\!] +_L \mathcal{O}[\![r_2]\!] \\
\mathcal{O}[\![\texttt{ProdL}\ r_1\ r_2]\!] &= \mathcal{O}[\![r_1]\!] \times_L \mathcal{O}[\![r_2]\!] \\
\mathcal{O}[\![\texttt{MapO}\ f\ r]\!] &= f^{-1}(\mathcal{O}[\![r]\!]) \\
\mathcal{O}[\![\texttt{ListL}\ r]\!] &= [\mathcal{O}[\![r]\!]] \\
\mathcal{O}[\![\texttt{BagO}\ r]\!] &= \langle \mathcal{O}[\![r]\!] \rangle \\
\mathcal{O}[\![\texttt{SetO}\ r]\!] &= \{\mathcal{O}[\![r]\!]\} \\
\mathcal{O}[\![\texttt{Inv}\ r]\!] &= \mathcal{O}[\![r]\!]^{-1}
\end{aligned}
$$

The *ordering relation denoted by a possibly infinite order representation* is

$$\mathcal{O}[\![r]\!] = \bigcup_{n \geq 0} \mathcal{O}[\![r|_n]\!].$$

□

Theorem 6.4 *Let r be an order representation over type* `T`*. Then $\mathcal{O}[\![r]\!]$ is an ordering relation over T.*

PROOF We have $\mathcal{O}[\![r|_n]\!] \sqsubseteq \mathcal{O}[\![r|_{n+1}]\!]$ for all $n \geq 0$, and $\bigcup_{n \geq 0} \mathcal{O}[\![r|_n]\!]$ is the supremum. □

The level-k approximations provide a finitary stratification of the pairs and the elements in the definition set of the ordering relation denoted by a order representation.

Definition 6.5 [Rank] Let $r \in$ `Order T`. Let $x, y \in T$, not necessarily distinct. The *rank of x and y under r* is defined as

$$\mathit{rank}_r(x, y) = \min\{n \mid (x, y) \in \mathcal{O}[\![r|_n]\!] \vee (y, x) \in \mathcal{O}[\![r|_n]\!]\}$$

with $\mathit{rank}_r(x, y) = \infty$ if $x \#_{\mathcal{O}[\![r]\!]} y$. Finally, define the *rank of x under r* by $\mathit{rank}_r(x) = \mathit{rank}_r(x, x)$. □

```
comp :: Order t → t → t → Ordering
comp (NatO n) x y = if 0 ≤ x && x ≤ n && 0 ≤ y && y ≤ n
                    then compare x y
                    else error "Argument out of range"
comp TrivO _ _ = EQ
comp (SumL r1 _) (Left x) (Left y) = comp r1 x y
comp (SumL _ _) (Left _) (Right _) = LT
comp (SumL _ _) (Right _) (Left _) = GT
comp (SumL _ r2) (Right x) (Right y) = comp r2 x y
comp (ProdL r1 r2) (x1, x2) (y1, y2) =
   case comp r1 x1 y1 of { LT → LT ;
                           EQ → comp r2 x2 y2 ;
                           GT → GT }
comp (MapO f r) x y = comp r (f x) (f y)
comp (BagO r) xs ys = comp (MapO (csort r) (listL r)) xs ys
comp (SetO r) xs ys = comp (MapO (cusort r) (listL r)) xs ys
comp (Inv r) x y = comp r y x

lte :: Order t → t → t → Bool
lte r x y = ordVal == LT || ordVal == EQ
    where ordVal = comp r x y

csort :: Order t → [t] → [t]
csort r = sortBy (comp r)

cusort :: Order t → [t] → [t]
cusort r = map head ∘ groupBy (lte (Inv r)) ∘ sortBy (comp r)
```

Figure 2: Generic comparison, sorting and unique-sorting functions

Observe that $rank_r(x,y) = rank_r(y,x)$; $rank_r(x,y) < \infty$ if and only if $(x,y) \in \mathcal{O}[\![r]\!] \vee (y,x) \in \mathcal{O}[\![r]\!]$; and $rank_r(x) < \infty$ if and only if $x \in \mathrm{def}(\mathcal{O}[\![r]\!])$. Note also that the rank of a pair not only depends on the ordering relation, but on the specific order representation to denote it.

Proposition 6.6 $rank_r(x,y) \leq \min\{rank_r(x), rank_r(y)\}$

PROOF If $(x,x) \in \mathcal{O}[\![r|_n]\!] \vee (y,y) \in \mathcal{O}[\![r|_n]\!]$ then $(x,y) \in \mathcal{O}[\![r|_n]\!] \vee (y,x) \in \mathcal{O}[\![r|_n]\!]$ by conditional comparability. Thus $rank_r(x,y) \leq rank_r(x)$ and $rank_r(x,y) \leq rank_r(y)$ by definition of rank. □

The level-k approximations allow us to treat order representations as if they were finite and prove results about them by structural induction. For example, consider the functions `comp`, `lte`, `csort` and `cusort` as defined in Figure 2. We can prove that `comp` implements the 3-valued comparison function, `lte` the Boolean version of `comp`, `csort` a sorting function, and

```
type Disc k = forall v. [(k, v)] → [[v]]

sdisc :: Order k → Disc k
sdisc _ []               = []
sdisc _ [(_, v)]         = [[v]]
sdisc (NatO n) xs         = sdiscNat n xs
sdisc TrivO xs            = [[ v | (_, v) ← xs ]]
sdisc (SumL r1 r2) xs    = sdisc r1 [ (k, v) | (Left k, v) ← xs ]
                         ++ sdisc r2 [ (k, v) | (Right k, v) ← xs ]
sdisc (ProdL r1 r2) xs   =
    [ vs | ys ← sdisc r1 [ (k1, (k2, v)) | ((k1, k2),  v) ← xs ],
           vs ← sdisc r2 ys ]
sdisc (MapO f r) xs      = sdisc r [ (f k, v) | (k, v) ← xs ]
sdisc (ListL r) xs       = sdisc (listL r) xs
sdisc (BagO r) xs        = sdiscColl updateBag r xs
    where updateBag vs v = v : vs
sdisc (SetO r) xs        = sdiscColl updateSet r xs
    where updateSet [] w = [w]
          updateSet vs@(v : _) w = if v == w then vs else w : vs
sdisc (Inv r) xs         = reverse (sdisc r xs)
```

Figure 3: Generic order discriminator `sdisc`

`cusort` a unique-sorting function, in each case for the order denoted by their respective first arguments. For `comp`, we specifically have:

Proposition 6.7 *For all order representations* r `:: Order T` *and* $x, y \in T$ *we have*

$$\texttt{comp}\ r\,x\,y = \begin{cases} \texttt{LT} & \text{if } x <_{\mathcal{O}[\![r]\!]} y \\ \texttt{EQ} & \text{if } x \equiv_{\mathcal{O}[\![r]\!]} y \\ \texttt{GT} & \text{if } x >_{\mathcal{O}[\![r]\!]} y \\ \bot & \text{if } x \#_{\mathcal{O}[\![r]\!]} y \end{cases}$$

PROOF (Idea) We can prove by induction on n that the 4 functions have the desired properties for all order representations $r|_n$; e.g., `comp` $r|_n\ x\ y =$ `EQ` $\Leftrightarrow x \equiv_{\mathcal{O}[\![r|_n]\!]} y$. This works since each of the functions, when applied to $r|_{n+1}$ on the left-hand side of a clause, is applied to order representation(s) $r'|_n$ on the respective right-hand side. From this the result follows for infinite r. □

7 Generic order discrimination

Having defined and illustrated an expressive language for specifying orders we are now in a position to define the generic order discriminator `sdisc`. See Figure 3.

```
sdiscNat :: Int → Disc Int
sdiscNat n xs = filter (not ∘ null) (bdiscNat n update xs)
   where update vs v = v : vs

bdiscNat :: Int → ([v] → v → [v]) → [(Int, v)] → [[v]]
bdiscNat (n :: Int) update xs =
    map reverse (elems (accumArray update [] (0, n-1) xs))
```

Figure 4: Bucket-sorting discriminator `sdiscNat`

The type

```
type Disc k = forall v. [(k, v)] → [[v]]
```

of a discriminator is polymorphic to capture its value parametricity property.

The clauses for the empty argument list, the trivial order, sum order, preimage and inverse are self-explanatory. The innocuous-looking clause

```
sdisc _ [(_, v)]          = [[v]]
```

is important for practical efficiency: A call to `sdisc` with a singleton input pair returns immediately *without* inspecting the key. This ensures that only distinguishing parts of the keys need to be inspected during execution. In the specific case of alphabetic string sorting, this implements the property of most-significant digit first (MSD) lexicographic sorting of only inspecting the minimum distinguishing prefix of its input.

7.1 Basic order discrimination

The clause

```
sdisc (NatO n) xs          = sdiscNat n xs
```

in the definition of `sdisc` invokes the *basic order discriminator* `sdiscNat` n for keys in the range $\{0, \ldots, n\}$. Our implementation of `sdiscNat` uses bucket sorting, presented in Figure 4. The function call `bdiscNat` n `update` $\vec{x}$ allocates a bucket table $T[0 \ldots n]$ and initializes each element $T[i]$ to the empty list. It then iterates over all $(k, v) \in \vec{x}$, appending v to the contents of $T[k]$. Finally, it returns the lists $T[k]$ in index order $k = 0 \ldots n$. Each list returned contains the values associated with the same k in the input. Since such lists may be empty, `sdiscNat` removes any empty lists. Traversing in index order ensures that groups of values associated with the same key are returned in ascending key order, as required of an order discriminator.

Apart from order representations involving `TrivO`, *all* calls to any order discriminator eventually result in—potentially many—leaf calls to `sdiscNat`. Thus the performance of `sdiscNat` is crucial for the performance of nearly *every* discriminator. Ours is a very simple implementation, but we emphasize that `sdisc` is essentially parameterized in `sdiscNat`: Dropping in any

high-performance implementation essentially bootstraps its performance via `sdisc` to order discrimination for arbitrary denotable ordering relations.

The code in Figure 4 implements the appending of a value to the contents of a table bucket by actually prepending it and eventually reversing it. We remark that eliding the final reversing of the elements of the array results in a *reverse stable* order discriminator. It can be checked that reverse stable discriminators can also be used in the remainder of the paper, saving the cost of list reversals. We shall stick to stable discriminators for clarity and simplicity, however.

7.2 Lexicographic product order discrimination

Consider now the clause

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

in Figure 3 for lexicographic product orders. First, each key-value pair is reshuffled to associate the second key component with the value originally associated with the key. Then the reshuffled pairs are discriminated on the first key component. This results in a list of groups of pairs, each consisting of a second key component and an associated value. Each such group is discriminated on the second key component, and the concatenation of all the resulting value groups is returned. Note how well the type of discriminators fits the compositional structure: We exploit the ability of the discriminator on the first key component to work with *any* associated values, and discarding the keys in the output of a discriminator makes the second key component discriminator immediately applicable to the output of the first key component discriminator.

7.3 Lexicographic list order discrimination

Lexicographic list order discrimination is implemented by order discrimination on the recursively defined order constructor `listL` in Section 6.3:

```
sdisc (ListL r) xs       = sdisc (listL r) xs
```

It is instructive to follow the execution of `sdisc (listL r)`, since it illustrates how an order representation functions a control structure for invoking the individual clauses of `sdisc`.

Example 7.1 Let us trace the execution of `sdisc ordString8` on input

$$\vec{x}_0 = [(\texttt{"bac"}, 1), (\texttt{"ac"}, 2), (\texttt{"cab"}, 3), (\texttt{""}, 4), (\texttt{"ac"}, 5)].$$

```
sdisc string8 x0 =
sdisc (ListL ordChar8) x0 =
sdisc (listL ordChar8) x0 =
sdisc (MapO fromList (SumL ordUnit (ProdL ordChar8 (listL ordChar8)))) x0 =
sdisc (SumL ordUnit (ProdL ordChar8 (listL ordChar8))) x1 =
sdisc ordUnit x2 ++ sdisc (ProdL ordChar8 (listL ordChar8)) x3
```

where

$$\begin{aligned}
\vec{x}_1 &= [(\texttt{Right ('b', "ac")}, 1), (\texttt{Right ('a', "c")}, 2),\\
&\quad\ (\texttt{Right ('c', "ab")}, 3), (\texttt{Left ()}, 4), (\texttt{Right ('a', "c")}, 5)]\\
\vec{x}_2 &= [(\texttt{()}, 4)]\\
\vec{x}_3 &= [((\texttt{'b', "ac"}), 1), ((\texttt{'a', "c"}), 2), ((\texttt{'c', "ab"}), 3), ((\texttt{'a', "c"}), 5)]
\end{aligned}$$

Since $\vec{x}_2$ is a singleton list, the second claus of `sdisc` yields

$$\texttt{sdisc ordUnit } \vec{x}_2 = [[4]].$$

Let us evaluate `sdisc (ProdL ordChar8 (listL ordChar8))` $\vec{x}_3$ then:

```
sdisc (ProdL ordChar8 (listL ordChar8)) x3 =
[ vs | ys <- sdisc ordChar8 x4, vs <- sdisc (listL ordChar8) ys] =
[ vs | ys <- sdisc (NatO 255) x5, vs <- sdisc (listL ordChar8) ys] =
[ vs | ys <- [[("c",2),("c",5)],[("ac",1)],[("ab",3)]],
       vs <- sdisc (listL ordChar8) ys] =
sdisc (listL ordChar8) [("c",2),("c",5)] ++
    sdisc (listL ordChar8) [("ac",1)] ++
    sdisc (listL ordChar8) [("ab",3)] =
sdisc (listL ordChar8) [("",2),("",5)] ++ [[1]] ++ [[3]] =
[[2,5]] ++ [[1]] ++ [[3]] =
[[2,5],[1],[3]]
```

where

$$\begin{aligned}
\vec{x}_4 &= [(\texttt{'b'}, (\texttt{"ac"}, 1)), (\texttt{'a'}, (\texttt{"c"}, 2)), (\texttt{'c'}, (\texttt{"ab"}, 3)), (\texttt{'a'}, (\texttt{"c"}, 5))]\\
\vec{x}_4 &= [(98, (\texttt{"ac"}, 1)), (97, (\texttt{"c"}, 2)), (99, (\texttt{"ab"}, 3)), (97, (\texttt{"c"}, 5))]
\end{aligned}$$

Putting everything together we have

$$\texttt{sdisc string8 } \vec{x} = [[4], [2,5], [1], [3]].$$

□

7.4 Bag and set order discrimination

The bag order $\langle R \rangle$ on lists can be implemented by sorting each list according to R and then applying the lexicographic order on the resulting lists. Consequently, if r denotes R, we can denote $\langle R \rangle$ by `bagO` r where

```
bagO r = MapO (csort r) (listL r)
```

and where `csort` is the generic comparison-based sorting function from Figure 2. This shows that, just like `ListL`, the order constructor `BagO` is redundant in the sense that it is definable using the other order constructors, and we could define

```
sdisc (BagO r) xs = sdisc (bagO r) xs
```

as we have done for the lexicographic list order `ListL`.

This typically[6] yields an $O(N \log N)$ algorithm, where N is the size of the input, for bag order discrimination.

We can do asymptotically better, however. The key insight is that, for the final lexicographic list discrimination step in bag order processing, we only need the *ordinal number* of an element of a key, not the element itself. This avoids reprocessing of the elements after sorting each of the keys.

Definition 7.2 [Ordinal number] Let R be an ordering relation and $K = [k_1, \ldots, k_n]$, $k_i \in \mathrm{def}(R)$ for all $i = 1, \ldots, n$. The ordinal number $\mathcal{N}_R^K(k_i)$ *of* k_i *under* R *within* K is the maximum number of pairwise R-inequivalent elements $k' \in K$ such that $k' <_R k_i$. □

Example 7.3

1. Let $K = [0, \ldots, n]$ for $n \geq 0$. Let $R = [n]$. Then $\mathcal{N}_R^K(k) = k$ for all $k \in \{0, \ldots, n\}$.
2. Let $K = [4, 9, 24, 11, 14]$ under the even-odd ordering O_{eo} in Example 4.4. Then the ordinal number of $4, 24$ and 14 is 0, and the ordinal number of 9 and 11 is 1.

□

Our discrimination algorithm for `BagO` r works as follows:

(1) Given input $[(\vec{k}_1, v_1), \ldots, (\vec{k}_n, v_n)]$, with $\vec{k}_i = [k_{i1}, \ldots, k_{im_i}]$, sort the $\vec{k}_i$ according to r, but return the ordinal numbers of their elements under r within $[k_{11}, \ldots, k_{1m_1}, \ldots k_{n1}, \ldots, k_{nm_n}]$, instead of the elements themselves.

(2) Perform lexicographic list order discrimination on `listL (NatO l)`, where l is the maximal ordinal number of any element in $\vec{k}_1 \ldots \vec{k}_n$ under r.

Step 1 is implemented efficiently as follows:

(a) Associate each key element k_{ij} with i, its *key index*.

(b) Discriminate the (key element, key index) pairs for r. This results in groups of key indices associated with $\equiv_r$-equivalent key elements, listed in ascending r-order. Observe that the j-th group in the result lists the indices of all the keys that contain a key element with ordinal number j. Let l be the maximal ordinal number of any key element.

[6] See Section 14 for the reason for the use of "typically" here.

```
sdiscColl :: ([Int] → Int → [Int]) → Order k → Disc [k]
sdiscColl update r xss = sdisc (listL (NatO (length keyNumBlocks - 1))) yss
     where
     (kss, vs)           = unzip xss
     elemKeyNumAssocs    = groupNum kss
     keyNumBlocks        = sdisc r elemKeyNumAssocs
     keyNumElemNumAssocs = groupNum keyNumBlocks
     sigs                = bdiscNat (length kss) update keyNumElemNumAssocs
     yss                 = zip sigs vs
```

Figure 5: Bag and set order discrimination

(c) Associate each key index with each of the ordinal numbers of its key elements.

(d) Discriminate the (key index, ordinal number) pairs for `NatO` l. This results in groups of ordinal numbers representing the key elements of the same key, but permuted into ascending order. We have to be careful to also return empty lists of ordinal numbers here, not just nonempty lists.[7] Since the groups are listed by key index, the groups of sorted ordinal numbers are listed in the same order as the keys $[\vec{k}_1, \ldots, \vec{k}_n]$ in the original input.

Figure 5 shows our implementation of `sdiscColl`, which abstracts the common steps for bag and set orders. For bag orders, `sdiscColl` is passed the function

```
updateBag vs v = v : vs
```

as its first argument. Set order discrimination is similar to bag order discrimination. The only difference is that we use

```
updateSet [] w = [w]
updateSet vs@(v : _) w = if v == w then vs else w : vs
```

instead of `updateBag`. The function `updateSet` eliminates duplicates in runs of identical ordinal numbers associated with the same key index in the computation of `sigs`. This is tantamount to unique-sorting the ordinal numbers of the elements of each key in the input.

Example 7.4 Let us trace the execution of `sdisc (BagO ordChar8)` on the input

$$\texttt{xss} = [(\texttt{"bac"}, 1), (\texttt{"ac"}, 2), (\texttt{"cab"}, 3), (\texttt{""}, 4), (\texttt{"ac"}, 5)].$$

from Example 7.1.

In `sdiscColl` we first unzip the value components from the keys:

[7] This was pointed out by an anonymous referee.

```
(kss, vs) = unzip xss
```

After this step we have

```
kss = ["bac", "ac", "cab", "", "ac"]
vs = [1, 2, 3, 4, 5]
```

(a) Next we perform a group numbering, which associates the key index with each of the element occurrences:

```
elemKeyNumAssocs    = groupNum kss
```

(Recall that `"bac"` is Haskell short-hand for `['b', 'a', 'c']`.) After this step we have

```
elemKeyNumAssocs = [('b', 0), ('a', 0), ('c', 0),
                    ('a', 1), ('c', 1),
                    ('c', 2), ('a', 2), ('b', 2),
                    ('a', 4), ('c', 4) ].
```

(b) We discriminate these pairs according to the key element ordering `ordChar8`:

```
keyNumBlocks        = sdisc ordChar8 elemKeyNumAssocs
```

which results in

```
keyNumBlocks = [ [0, 1, 2, 4], [0, 2], [0, 1, 2, 4] ]
```

in our example. The first group corresponds to key character `'a'`, the second to `'b'` and the third to `'c'`. The elements of each group are the indices, numbered $0, \ldots, 4$, of keys, in which a member of the particular equivalence class occurs; e.g. 0 is the index of `"bac"` and 2 of `"cab"`. So the group `[0, 2]` in `keyNumBlocks` expresses that the equivalence class represented by that group (the character `'b'`) occurs once in the key with index 0 (`"bac"`) and once in the key with index 2 (`"cab"`), and in no other keys. Note that 3 does not occur in `keyNumBlocks` at all, since the key with index 3 is empty.

(c) Next we convert `keyNumBlocks` into its group number representation:

```
keyNumElemNumAssocs = groupNum keyNumBlocks,
```

which results in the binding

```
keyNumElemAssocs = [ (0, 0), (1, 0), (2, 0), (4, 0),
                     (0, 1), (2, 1),
                     (0, 2), (1, 2), (2, 2), (4, 2) ].
```

Each pair (i, j) represents an element containment relation: the key with index i contains an element with ordinal number j. E.g., the pair `(4, 0)` expresses that the key with index `4`, the second occurrence of `"ac"`, contains an element with ordinal number `0`, the character `'a'`.

(d) We now discriminate these membership pairs:

```
sigs          = bdiscNat 5 updateBag keyNumElemNumAssocs
```

This collects together all the characters, represented by their ordinal numbers, that are associated with the same key. Each group thus represents a key from the input, but with each character replaced by its ordinal number. Using `bdiscNat` ensures that the groups are returned in the same order as the keys in `kss` and that empty value lists are returned, too. Since `bdisc` is stable, it returns the ordinal numbers in ascending order in each group. The resulting groups of ordinal numbers in our example are

```
sigs = [ [0, 1, 2], [0, 2], [0, 1, 2], [], [0, 2] ].
```

Observe that they represent the original keys `kss`, but each key ordered alphabetically into

```
["abc", "ac", "abc", "", "ac"]
```

and with ordinal numbers replacing the corresponding key elements.

Finally, we zip `sigs` with the value components `vs` from the original `xss`:

```
yss = zip sigs vs.
```

This gives

```
yss = [( [0, 1, 2], 1), ( [0, 2], 2), ( [0, 1, 2], 3), ([], 4), ([0, 2], 5) ]
```

Applying the list order discriminator

```
sdisc (listL (NatO (length keyNumBlocks - 1))) yss
```

where `length keyNumBlocks - 1` $= 2$, the final output is

```
[ [4], [1, 3], [2, 5] ],
```

as desired. □

Observe how bag and set order discrimination involves a discrimination step on key elements, which may result in recursive discrimination of nodes inside those elements, and two other discrimination steps on key indices and lists of ordinal numbers, respectively, which do *not* recurse into the keys.

7.5 Correctness

Theorem 7.5 *For each order representation* r `:: Order T`, `sdisc` r *is a stable order discriminator for* $\mathcal{O}[\![r]\!]$ *over* T.

PROOF (Sketch) By induction on $n = \max\{rank_r(k_i) \mid i \in \{1, \ldots, n\}\}$, where $\vec{x} = [(k_1, v_1), \ldots, (k_n, v_n)]$ is the input to `sdisc` r. The case for rank 0 is vacuously true. For the inductive case, we inspect each clause of `sdisc` in turn. In each case, the maximum rank of keys in a call to `sdisc` on the right-hand side is properly less than the maximum rank of the keys in the call on the left-hand side, which allows us to invoke the induction hypothesis, and we can verify that the values in the result are grouped as required of a stable order discriminator for $\mathcal{O}[\![r]\!]$. □

8 Complexity

In this section we prove that `sdisc` from Figure 3 typically produces worst-case linear-time order discriminators. In particular, it does so for the standard ordering relations on all regular recursive first-order types and thus yields linear-time partitioning and sorting algorithms for each.

Our machine model is a unit-cost random access machine (RAM) (Tarjan 1983) with fixed word width, where values are stored in fully boxed representation. It has basic instructions 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. Unit-cost means that pointer operations and operations on "small" integers—integer values polynomially bounded by the size of the input—take constant time. Random access means that array lookups using small integers as indices also takes constant time. Fixed word width means that the number of bits per word in RAM memory is constant (think 32 or 64). In particular, it does not change depending on the size of the input.[8]

We define the size of a value as follows.

Definition 8.1 [Size] The *(tree) size* of a value is defined as follows:

$$\begin{aligned} |c| &= 1 \\ |()| &= 1 \\ |inl\, v| &= 1 + |v| \\ |inr\, w| &= 1 + |w| \\ |(v, w)| &= 1 + |v| + |w| \\ |fold(v)| &= 1 + |v| \end{aligned}$$

□

[8] It might seem strange to allow dynamic word sizes. Such RAMs are, somewhat curiously, not uncommon in algorithms papers, however.

Note that the size function for pairs adds the size of each component separately. This means that the size function measures the storage requirements of an *unshared* (*unboxed* or *tree-structured*) representation 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 $O(n)$. The *top-down* (purely recursive) method embodied in our generic discriminators in this paper gives asymptotically optimal performance only for *unshared* data. Dealing with sharing efficiently requires *bottom-up discrimination* (Paige 1991; Henglein 2003), which builds upon top-down discrimination. Generic bottom-up discrimination is future work.

We write $\mathcal{T}_f(v)$ for the number of steps function f takes on input v.[9]

Definition 8.2 The set $\mathcal{L}$ of *linear-time discriminable* order representations is the set of all order representations r such that

$$\mathcal{T}_{\texttt{sdisc}\ r}([(k_1, v_1), \ldots, (k_n, v_n)]) = O(n + \textstyle\sum_{i=1}^{n} |k_i|). \qquad \square$$

8.1 Nonrecursive orders

The question now is: Which order representations are linear-time discriminable? Clearly, a function f must execute in linear time if the discriminator for `MapO` f r is to do so, too. Interestingly this is sufficient to guarantee that each *finite* order representation yields a linear-time discriminator.

Proposition 8.3 *Let r be a finite order representation, where each function occurring in r executes in linear time and produces an output of size linear in its input. Then r is linear-time discriminable.*

PROOF By structural induction on r. The key property is that a linear-time executable function f used as an argument to `MapO` in r can only increase the size of its output by a constant factor relative to the size of its input. Note that the output size limitation does not follow from f executing in linear time since it may produce a shared data structure with exponentially larger tree size. $\square$

It is important to note that the constant factor in the running time of `sdisc` r generally depends on r. So this result does not immediately generalize to order representations for recursive types.

8.2 Recursive orders

To get a sense of when an infinite order representation yields a linear-time order discriminator, let us investigate a situation where this does *not* hold.

Consider the order constructor `flipflop`

[9] Here, we use "function" in the sense of *code* implementing a mathematical function.

```
flipflop :: Order t → Order [t]
flipflop r = MapO (fromList ∘ reverse)
             (SumL ordUnit (ProdL r (flipflop r)))
```

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. Applying `sdisc` to `flipflop ordChar8` yields a quadratic time discriminator. The reason for this is the repeated application of the `reverse` function. We can observe that also the comparison function `comp (flipflop ordChar8)` takes quadratic time.

Let us look at the body of `flipflop` in more detail: We have an order representation r which satisfies

$$r' = \texttt{MapO (fromList . reverse) (SumL ordUnit (ProdL r } r' \texttt{))}.$$

Executing `sdisc` r' causes `sdisc` r' to be executed recursively. The reason for nonlinearity is that the recursive call operates on parts of the input that are also processed by the nonrecursive code, specifically by the `reverse` function.

The key idea to ensuring linear-time performance of recursive discriminators is the following: Make sure that the input can be (conceptually) split such that the execution of the body of `sdisc` r' *minus* its recursive calls to the *same* discriminator `sdisc` r' can be *charged* to one part of the input, and its recursive call(s) to the *other* part. Charging means that we attribute a constant amount of computation to constant amounts of the original input. 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 the pointers to those parts, but must not dereference them. How can we ensure that this is obeyed? We insist that the nonrecursive computation steps of `sdisc` r' only manipulate pointers to the parts passed to the recursive calls of `sdisc` r' without dereferencing or duplicating them. Intuitively, the nonrecursive code must be *parametric polymorphic* in the original sense of Strachey (2000)!

The main technical complication is extending this idea to order representations containing `MapO`. the presence of ensuring this as We do this by, conceptually, splitting the input keys, viewed from their roots, into top-level parts, which are are processed nonrecursively, and bottom-level parts, which are passed to the recursive call(s).

To formalize this splitting idea we extend types and order representations with formal type variables `t1, t2, ..., tn` and order variables `r1, r2, ..., rn` respectively. For simplicity, we restrict ourselves to adding a *single* type variable `t1` and a *single* order variable `r1` of type `Order t1` here:

Definition 8.4 Let `t1` be a distinct *type variable* and `r1` a formal *order variable*.

Then the *types* $\mathcal{T}^\infty[\texttt{t1}]$ *over* `t1` are the set of possibly infinite labeled trees built from the signature

$$\{A^{(0)}, \times^{(2)}, +^{(2)}, 1^{(0)}, \mathit{fold}^{(1)}, \texttt{t1}^0\}.$$

$\mathcal{R}^\infty[\texttt{r1}]$ is the set of typed labeled trees built from the constructors in Definition 6.1 with an additional formal constructor `r1 :: Order t1`. Furthermore, each f occurring in $R \in \mathcal{R}^\infty[\texttt{r1}]$ must have polymorphic type $\forall \texttt{t1}.T_1 \to T_2$ for some $T_1, T_2 \in \mathcal{T}^\infty[\texttt{t1}]$. □

We can now split the size of a value on type $\texttt{T} \in \mathcal{T}^\infty[t_1]$ into upper and lower parts.

Definition 8.5 [Upper and lower sizes] Let $T \in \mathcal{T}^\infty[\texttt{t1}]$. The *lower and upper sizes* $|.|_T$, respectively $|.|^T$, are defined as follows:

$$\begin{aligned}
|v|^{\texttt{t1}} &= 0 \\
|c|^A &= 1 \\
|()|^1 &= 1 \\
|\mathit{inl}\, v|^{T_1+T_2} &= 1 + |v|^{T_1} \\
|\mathit{inr}\, w|^{T_1+T_2} &= 1 + |w|^{T_2} \\
|(v,w)|^{T_1\times T_2} &= 1 + |v|^{T_1} + |w|^{T_2} \\
|\mathit{fold}(v)|^{\mu t.T} &= 1 + |v|^{T[(\mu t.T)/t]}
\end{aligned}$$

$$\begin{aligned}
|v|_{\texttt{t1}} &= |v| \\
|c|_A &= 0 \\
|()|_1 &= 0 \\
|\mathit{inl}\, v|_{T_1+T_2} &= |v|_{T_1} \\
|\mathit{inr}\, w|_{T_1+T_2} &= |w|_{T_2} \\
|(v,w)|_{T_1\times T_2} &= |v|_{T_1} + |w|_{T_2} \\
|\mathit{fold}(v)|_{\mu t.T} &= |v|_{T[(\mu t.T)/t]}
\end{aligned}$$

□

Proposition 8.6 *For all values v and types $T \in \mathcal{T}^\infty[\texttt{t1}]$ we have $|v| = |v|^T + |v|_T$ whenever both sides are defined.*

PROOF By complete (course of values) induction on $|v|$. □

The key property for proving linear-time discriminability for infinite order representations is that polymorphic functions occurring in `MapO` order representations must be linear-time computable in a strong sense:

Definition 8.7 [Strongly linear-time computable function] We say a function $f :: \forall \mathtt{t1}.T_1 \to T_2$ is *strongly linear-time computable* if

1. $\mathcal{T}_f(k) = O(|k|^{T_1})$.
2. $|f(k)|^{T_2} = O(|k|^{T_1})$.
3. $|f(k)|_{T_2} \leq |k|_{T_1}$.

□

Note that the last condition is without O.
Here are some examples of linear-time computable functions:

- The identity function $id :: \forall \mathtt{t1}.\mathtt{t1} \to \mathtt{t1}$.
- The list length function $length :: \forall \mathtt{t1}.[\mathtt{t1}] \to \mathsf{Int}$.
- The list reverse function $reverse :: \forall \mathtt{t1}.[\mathtt{t1}] \to [\mathtt{t1}]$.

The argument duplication function $dup :: \forall \mathtt{t1}.\mathtt{t1} \to \mathtt{t1} \times \mathtt{t1}$, on the other hand, is not linear-time computable: it violates the third condition in Definition 8.7.

Since we measure the *tree size* of values, a function can produce outputs of asymptotically larger size than its running time due to sharing. Consider the function $repFstElem :: \forall t.[t] \to [t]$, which takes as input $[v_1, \ldots, v_n]$ and returns $[\overbrace{v_1, \ldots, v_1}^{n}]$ for $n \geq 1$. Applying it to a list with a first element of size m, followed by m elements of size 1 yields a result of size $\Theta(m^2)$. It satisfies Property 1 but not Property 2 (nor Property 3 for that matter). This shows that Property 1 of Definition 8.7 does not imply Property 2.

We can now give a recipe for constructing order representations over recursive types that yield linear-time discriminators:

1. Let $T = \mu \mathtt{t1}.T'$ be a recursive type with $f : T \to T'[T/\mathtt{t1}], f(fold(v)) = v$ the unfold-part of the isomorphism between T and $T'[T/\mathtt{t1}]$.
2. Find a finite order representation $r' :: \mathtt{Order}\ T'$ containing only strongly linear-time computable functions.
3. Define $r :: \mathtt{Order}\ T$ recursively by $r = \mathtt{MapO}\ f\, r'[r/\mathtt{r1}]$.

Then R is linear-time discriminable. We sketch a proof of this below.

Definition 8.8 [$\mathcal{T}^1$] Define $\mathcal{T}^1_{\mathtt{sdisc}\, r'[r/\mathtt{r1}]}(\vec{x})$ to be the execution time of $\mathtt{sdisc}\, r'[r/\mathtt{r1}](\vec{x})$, but not counting any calls of the form $\mathtt{sdisc}\, r\, (\vec{y})$. □

Lemma 8.9 *Let $T = \mu \mathtt{t1}.T'$. Let $r :: \mathtt{Order}\ T$, $r' :: \mathtt{Order}\ T'$ finite, and let all functions f occurring in r' be strongly linear-time computable. Then*

1. $\mathcal{T}^1_{\texttt{sdisc}\, r'[r/\texttt{r1}]}(\vec{x}) = O(n + \sum_{i=1}^{n} |k_i|^{T'})$ *where* $\vec{x} = [(k_1, v_1), \ldots, (k_n, v_n)]$.

2. *The bag of calls* $\langle \texttt{sdisc}\, r\, (\vec{z}_j) \rangle_j$ *invoked during execution of* $\texttt{sdisc}\, r'[r/\texttt{r1}](\vec{x})$ *has the property that* $\sum_j |\vec{z}_j| \leq \sum_{i=1}^{n} |k_i|_{T'}$.

PROOF (Sketch) The proof is by structural induction on r'. The most interesting cases are `MapO` $f\, r''$, `ListL` r'', `BagO` r'', and `SetO` r''.

- For `MapO` $f\, r''$ the requirement of strong linear-time computability of f is sufficient to make the induction step go through.

- For `ListL` r'', consider the recursive applications of `sdisc` during evaluation of `sdisc (ListL` r''`)` $\vec{x}$. Let us *charge* the *nonrecursive* computation steps of a call to `sdisc` r''' (for any r''') to the *roots* (only!) of the keys in the input. (Recall that we assume a fully-boxed data representation. The space requirement of each node of such a representation is accounted for by the additive constant 1 in Definition 8.1.) It is straightforward to check that each node is then charged with a constant number of computation steps, since each node occurs *at most once* as the root of a key in the input of a call to `sdisc`r''' for some r''' during the evaluation of `sdisc (ListL` r''`)` $\vec{x}$.

- For `BagO` r'', part 1 of the lemma follows from the fact that, by definition, `sdisc (BagO r)` consists of: one invocation of `sdisc r`, which, inductively, executes in linear time in the aggregate size of the key elements of the input; and the remaining steps, which are linear in the size of the *remaining nodes* in the input. For Part 2 of the lemma it is important that only the call to `sdisc r` operates on key elements, and the final call `sdisc (ListL ...) yss` is on the ordinal numbers of the key elements, not the key elements themselves.

- For `SetO` r'' the argument is the same as for `BagO` r''.

□

We can now apply Lemma 8.9 recursively.

Theorem 8.10 *Let* $T = \mu\texttt{t1}.T'$ *with* $f :: T \to T'[T/\texttt{t1}], f(fold(v)) = v$, *the unfold-function from* T. *Let* $r ::$ `Order` T *and finite* $r' ::$ `Order` T' *such that*

$$r = \texttt{MapO}\ f\, (r'[r/\texttt{r1}]).$$

Furthermore, let all functions occurring in r' *be strongly linear-time computable.*

Then r *is linear-time discriminable.*

PROOF Consider $\texttt{sdisc}\, r\, (\vec{x})$ where $\vec{y} = [(f(k), v) | (k, v) \in \vec{x}]$.

$$\begin{aligned}
\mathcal{T}_{\texttt{sdisc}\, r}(\vec{x}) &= \mathcal{T}_{\texttt{sdisc}\, \texttt{Map0}\, f\, (r'[r/\texttt{r1}])}(\vec{x}) \\
&= \mathcal{T}_{\texttt{sdisc}\, r'[r/\texttt{r1}]}(\vec{y}) + O(\sum_{i=1}^{n} |k_i|^{T'}) \text{ by properties of } f \\
&= \mathcal{T}^1_{\texttt{sdisc}\, r'[r/\texttt{r1}]}(\vec{y}) + O(\sum_{i=1}^{n} |k_i|^{T'}) + \text{ all recursive calls to } \texttt{sdisc}\, r \\
&= O(\sum_{i=1}^{n} |k_i|^{T'}) + O(\sum_{i=1}^{n} |k_i|^{T'}) + \sum_j \mathcal{T}_{\texttt{sdisc}\, r}(\vec{z}_j) \\
&= O(\sum_{i=1}^{n} |k_i|^{T'}) + \sum_j \mathcal{T}_{\texttt{sdisc}\, r}(\vec{z}_j)
\end{aligned}$$

where $\sum_j |\vec{z}_j| \leq \sum_{i=1}^{n} |k_i|_{T'}$ by Lemma 8.9. Since $|\vec{x}| \geq \sum_{i=1}^{n} |k_i| = \sum_{i=1}^{n} |k_i|^{T'} + \sum_{i=1}^{n} |k_i|_{T'}$ we can see that the number of the execution steps excepting the recursive ones to $\texttt{sdisc}\, r$ is linear bounded by one part of the input, and all the recursive calls of $\texttt{sdisc}\, r$ can be attributed to the other part of the input, with the same constant factor. Consequently, the whole execution is linear bounded in the size of the keys in the input, and thus $\texttt{sdisc}\ \ r$ is linear-time discriminable. □

Each regular recursive type T has a standard order r_T denoted by a canonical order representation: product types are ordered by `ProdL`, sum types by `SumL`, `Int` by its standard order, `t1` by `r1`, and a recursive type $T = \mu\texttt{t1}.\, T'$ by $r = \texttt{Map0}\ \ f\ (r'[r/\texttt{r1}])$ where r' is the canonical order representation for T' and f is the unfold function from T to $T'[T/\texttt{t1}]$.

Corollary 8.11 *Let T be a regular recursive first-order type. Then r_T, the canonical order representation for T, is linear-time discriminable.*

PROOF The conditions of Theorem 8.10 are satisfied. □

We have observed that whenever a discriminator is superlinear, so is the comparison function. We conjecture that `sdisc` has the same asymptotic behavior as the generic binary comparison function `comp` (see Figure 2).

Conjecture 8.12 *Let $\mathcal{T}'_{\texttt{comp}\, r}(n) = \max\{\mathcal{T}_{\texttt{comp}\, r}(x_1)(x_2) \mid |x_1| + |x_2| \leq n\}$ and $\mathcal{T}'_{\texttt{sdisc}\, r}(n) = \max\{\mathcal{T}_{\texttt{sdisc}\, r}([(k_1, v_1), \ldots, (k_m, v_m)]) \mid \sum_{i=1}^{m} |k_i| \leq n\}$. Then $\mathcal{T}'_{\texttt{sdisc}\, r} = O(\mathcal{T}_{\texttt{comp}\, r})$.* □

The conjecture expresses that discriminators are a proper generalization of the corresponding comparison functions for all R, not just the linear-time discriminable: They asymptotically execute within the same computational resource bounds, but decide the ordering relation on m arguments (of aggregate size n) instead of just 2 arguments (of combined size n).

```
data Equiv t where
     NatE     :: Int → Equiv Int
     TrivE    :: Equiv t
     SumE     :: Equiv t1 → Equiv t2 → Equiv (Either t1 t2)
     ProdE    :: Equiv t1 → Equiv t2 → Equiv (t1, t2)
     MapE     :: (t1 → t2) → Equiv t2 → Equiv t1
     ListE    :: Equiv t → Equiv [t]
     BagE     :: Equiv t → Equiv [t]
     SetE     :: Equiv t → Equiv [t]
```

Figure 6: Equivalence representations

9 Equivalence representations

In the previous sections we have seen how to implement order discrimination efficiently by structural recursion over order representations. In this section we shall do the same for *equivalences*. The presentation is condensed where the techniques are essentially the same as for order discrimination. We emphasize that the practical benefits of equivalence discrimination are most pronounced for references, which have no natural ordering relation, and for problems where the output is not required to be ordered.

9.1 Equivalence constructors

As for ordering relations (Section 6) there are common constructions on equivalence relations. The following are equivalence relations:

- The *empty relation* $\emptyset$, on any set S.
- The *trivial relation* $S \times S$, on S.
- For each nonnegative n, the identity relation $\equiv_{[n]}$ on $\{0, \ldots n\}$.

Given $E_1 \in \mathit{Equiv}(T_1), E_2 \in \mathit{Equiv}(T_2), f \in T_1 \to T_2$, the following are also equivalence relations:

- The *sum equivalence* $E_1 +_E E_2$, over $T_1 + T_2$, defined by

$$x \equiv_{E_1 +_E E_2} y \Leftrightarrow \begin{cases} (x = \mathit{inl}\, x_1 \wedge y = \mathit{inl}\, y_1 \wedge x_1 \equiv_{E_1} y_1) \vee \\ (x = \mathit{inr}\, x_2 \wedge y = \mathit{inr}\, y_2 \wedge x_2 \equiv_{E_2} y_2) \\ \text{for some } x_1, y_1 \in T_1, x_2, y_2 \in T_2. \end{cases}$$

- The *product equivalence* $E_1 \times_E E_2$, over $T_1 \times T_2$, defined by

$$(x_1, x_2) \equiv_{E_1 \times_E E_2} (y_1, y_2) \Leftrightarrow x_1 \equiv_{E_1} y_1 \wedge x_2 \equiv_{E_2} y_2.$$

```
eq :: Equiv t → t → t → Bool
eq (NatE n) x y = if 0 ≤ x && x ≤ n && 0 ≤ y && y ≤ n
                  then (x == y)
                  else error "Argument out of range"
eq TrivE _ _ = True
eq (SumE e1 _) (Left x) (Left y) = eq e1 x y
eq (SumE _ _) (Left _) (Right _) = False
eq (SumE _ _) (Right _) (Left _) = False
eq (SumE _ e2) (Right x) (Right y) = eq e2 x y
eq (ProdE e1 e2) (x1, x2) (y1, y2) =
   eq e1 x1 y1 && eq e2 x2 y2
eq (MapE f e) x y = eq e (f x) (f y)
eq (ListE e) xs ys = eq (listE e) xs ys
eq (BagE _) [] [] = True
eq (BagE _) [] (_ : _) = False
eq (BagE e) (x : xs') ys =
   case delete e x ys of Just ys' → eq (BagE e) xs' ys'
                         Nothing → False
   where
    delete :: Equiv t → t → [t] → Maybe [t]
    delete e v = subtract' []
       where subtract' _ [] = Nothing
             subtract' accum (x : xs) =
                if eq e x v then Just (accum ++ xs)
                            else subtract' (x : accum) xs
eq (SetE e) xs ys =
  all (member e xs) ys && all (member e ys) xs
  where member :: Equiv t → [t] → t → Bool
        member _ [] _ = False
        member e (x : xs) v = eq e v x || member e xs v
```

Figure 7: Generic equivalence test

- The *preimage* $f^{-1}(E_2)$ of E_2 under f, over T_1, defined by

$$x \equiv_{f^{-1}(E_2)} y \Leftrightarrow f(x) \equiv_{E_2} f(y).$$

- The *list equivalence* $\equiv_{[E_1]}$, also written E_1^*, over T_1^*, defined by

$$[x_1, \ldots, x_m] \equiv_{[E_1]} [y_1, \ldots, y_n] \Leftrightarrow \\ m = n \wedge \forall 1 \leq j \leq m.\ x_j \equiv_{E_1} y_j$$

- The *bag equivalence* $\equiv_{\langle E_1 \rangle}$ on T_1^*, over T_1^*, defined by

$$\vec{x} \equiv_{\langle E_1 \rangle} \vec{y} \Leftrightarrow \exists x'.\vec{x} \cong \vec{x}' \wedge \vec{x}' \equiv_{[E_1]} \vec{y}.$$

  (Recall that $\vec{x} \cong \vec{x}'$ means that $\vec{x}'$ is permutation of $\vec{x}$.)

- The *set equivalence* $\{E_1\}$ on T_1^*, over T_1^*, defined by

$$\vec{x} \equiv_{\{E_1\}} \vec{y} \Leftrightarrow (\forall i.\exists j.\ x_i \equiv_{E_1} y_j) \wedge (\forall j.\exists i.\ x_i \equiv_{E_1} y_j).$$

Treating the equivalence constructions as constructors, we can define *equivalence representations* the same way we have done for order representations. See Figure 6. Using domain-theoretic arguments as for order representations (Theorems 5.1 and 6.4), each equivalence representation e, whether finite or infinite, denotes an equivalence relation $\mathcal{E}[\![e]\!]$.

Theorem 9.1 *Let e be an equivalence representation. Then $\mathcal{E}[\![e]\!]$ is an equivalence relation.*

Analogous to Proposition 6.7 it is possible to characterize $\mathcal{E}[\![e]\!]$ by the generic equivalence testing function `eq :: Equiv t -> t -> t -> Bool` in Figure 7.

Proposition 9.2 *For all equivalence representations e over T, $x, y \in T$*

$$\texttt{eq}\ e\, x\, y = \begin{cases} \texttt{True} & \text{if } x \equiv_{\mathcal{O}[\![e]\!]} y \\ \texttt{False} & \text{if } x \not\equiv_{\mathcal{O}[\![e]\!]} y \wedge (x \in \mathrm{def}(e) \vee y \in \mathrm{def}(e)) \\ \bot & \text{if } x \not\equiv_{\mathcal{O}[\![e]\!]} y \wedge x \notin \mathrm{def}(e) \wedge x \notin \mathrm{def}(e) \end{cases}$$

9.2 Definable equivalence constructors

We can denote the identity relations (equality) on basic types:

```
eqUnit :: Equiv ()
eqUnit = TrivE

eqBool :: Equiv Bool
eqBool = MapE bool2sum (SumE eqUnit eqUnit)
  where bool2sum :: Bool → Either () ()
```

```
        bool2sum False = Left ()
        bool2sum True = Right ()

eqNat8 :: Equiv Int
eqNat8 = NatE 255

eqNat16 :: Equiv Int
eqNat16 = NatE 65535

eqInt32 :: Equiv Int
eqInt32 = MapE splitW (ProdE eqNat16 eqNat16)

eqChar8 :: Equiv Char
eqChar8 = MapE ord eqNat8

eqChar16 :: Equiv Char
eqChar16 = MapE ord eqNat16
```

Observe how equality representation `eqInt32` on 32-bit integers is defined in what appears to be a rather roundabout fashion: It splits integers into their upper and lower 16-bits and then performs equality on these pairs componentwise as unsigned 16-bit integers. (The function `splitW` is defined in Section 6.2.) The reason for this is as before: To enable efficient basic discrimination by using a bucket array indexed by 16-bit integers. This can also be done using 8, 24, or any other number of bits, or any combination thereof, but we shall restrict ourselves to 16-bit indexed arrays for simplicity.

The general recipe for defining equivalence representations on recursive types is the same as for order representations in Section 6.4. In particular, list equivalence is definable as follows:

```
listE :: Equiv t → Equiv [t]
listE e = MapE fromList (SumE eqUnit (ProdE e (listE e)))
```

where `fromList` is as in Section 6.3. Using `listE` we can define string equality:

```
eqString8 :: Equiv String
eqString8 = list eqChar8
```

10 Generic equivalence discrimination

We can now give the complete definition of the generic equivalence discriminator `disc`, which is indexed by equivalence representations; see Figure 8. Let us look at the main differences to `sdisc`.

10.1 Basic equivalence discrimination

A *basic equivalence discriminator* is like the bucket-sorting based order discriminator `sdiscNat` n from Figure 4, with the exception that it returns the

```
disc :: Equiv k → Disc k
disc _ []               = []
disc _ [(_, v)]         = [[v]]
disc (NatE n) xs        =
   if n < 65536 then discNat16 xs else disc eqInt32 xs
disc TrivE xs           = [map snd xs]
disc (SumE e1 e2) xs    = disc e1 [ (k, v) | (Left k, v) ← xs ] ++
                          disc e2 [ (k, v) | (Right k, v) ← xs ]
disc (ProdE e1 e2) xs   =
   [ vs | ys ←disc e1 [ (k1, (k2, v)) | ((k1, k2), v) ← xs ],
          vs ← disc e2 ys ]
disc (MapE f e) xs      = disc e [ (f k, v) | (k, v) ← xs ]
disc (ListE e) xs       = disc (listE e) xs
disc (BagE e) xs        = discColl updateBag e xs
disc (SetE e) xs         = discColl updateSet e xs
```

Figure 8: Generic equivalence discriminator `disc`

groups in the order the keys occur in the input, instead of ordered numerically. It can be implemented as follows. When applied to key-value pairs $\vec{x}$:

(1) Allocate a bucket table $T[0 \ldots n]$ and initialize each bucket to the empty list. Allocate a variable K for holding key lists, also initialized to the empty list.

(2) Iterate over all $(k, v) \in \vec{x}$, appending v to $T[k]$, and, if k is encountered for the first time, append k to K.

(3) Iterate over all keys $k \in K$, outputting $T[k]$.

Figure 9 shows an implementation in Haskell using the ST monad, which allows encapsulating the imperative updates to a locally allocated array as an observably side-effect free function. Even though the final index order traversal is avoided, it still suffers from the same deficit as `sdiscNat`: Every application `discNatST` n $\vec{x}$ results in the allocation and complete initialization of a bucket table $T[0 \ldots n]$.

Paige and Tarjan (1987) employ the array initialization trick of Aho et al. (1983) to get around complete table initialization. We can go one step further: Avoid allocation of a new bucket table for each call altogether. The key idea is to use a *global* bucket table $T[0 \ldots n]$, whose elements are guaranteed to be empty lists before and after a call to the basic equivalence discriminator.

We define a function `discNat`, which generates efficient basic equivalence discriminators. A call to `discNat` n does the following:

```
discNatST :: Int → Disc Int
discNatST n xs =
  runST (
    do table ← newArray (0, n) [] :: ST s (STArray s Int [v]) ;
       ks ← foldM (λkeys (k, v) →
                do vs ← readArray table k ;
                   case vs of [] → do writeArray table k [v] ;
                                      return (k : keys) ;
                              _  → do writeArray table k (v : vs) ;
                                      return keys )
                   [] xs ;
       foldM (λvss k → do elems ← readArray table k ;
                          return (reverse elems : vss) )
             [] ks
   )
```

Figure 9: Basic equivalence discriminator, implemented using ST monad *Not used—too inefficient!*

(1) Allocate a bucket table $T[0 \ldots n]$ and initialize each element to the empty list.

(2) Return a function that, when passed key-value pairs $\vec{x}$, executes the following:

   (a) Allocate a variable K for a list of keys, initialized to the empty list.

   (b) Iterate over all $(k, v) \in \vec{x}$, appending v to $T[k]$ and, if k is encountered for the first time, appending k to K.

   (c) Iterate over all keys $k \in K$, outputting $T[k]$ *and resetting* $T[k]$ *to the empty list.*

Note that executing `discNat` n allocates a bucket table and returns a function, where each call reuses the *same* bucket table. The function requires that the bucket table contain only empty lists before executing the first step above (2a); it reestablishes this invariant in the final step (2c). The upshot is that the function does not allocate a new table every time it is called and executes in time $O(|\vec{x}|)$, independent of n, instead of $O(|\vec{x}| + n)$, which is critical for practical performance.

The basic discriminator returned by a call to `discNat` n is neither reentrant nor thread-safe nor resilient to exceptions thrown during its execution due to the possibility of unsynchronized accesses and imperative updates to the bucket table shared by all calls. Consequently, each thread should use a basic discriminator with a thread-local bucket table, and, in a language with lazy evaluation such as Haskell, all keys in the input should be fully

```
discNat :: Int → Disc Int
discNat n =
  unsafePerformIO (
  do { table ← newArray (0, n) [] :: IO (IOArray Int [v]) ;
       let discNat' xs = unsafePerformIO (
            do { ks ← foldM (λkeys (k, v) →
                        do { vs ← readArray table k ;
                             case vs of {
                               [] → do { writeArray table k [v] ;
                                         return (k : keys) } ;
                               _ →  do { writeArray table k (v : vs) ;
                                         return keys } } } )
                      [] xs ;
                foldM (λvss k → do { elems ← readArray table k ;
                                     writeArray table k [] ;
                                     return (reverse elems : vss) })
                      [] ks } )
       in return discNat' } )
```

Figure 10: Basic equivalence discriminator generator `discNat`

evaluated before the first key is stored in the bucket table. If the basic discriminator is used for discriminating references implemented by raw machine addresses, garbage collection needs to be carefully synchronized with calls to it. Finally, the shared imperative use of a bucket table in multiple calls makes sound typing of the basic discriminator in Haskell or other currently employed type systems impossible. In Haskell, it rules out the use of the ST monad to give a purely functional type to the basic discriminator returned by `discNat` n. For these reasons and the central role they play in practically efficient discrimination, sorting and partitioning, we believe basic discriminators for 8-, 16-, 32- and 64-bit words should be built into statically typed functional programming languages as primitives, analogous to comparison functions being built-in.

For experimentation, we provide an implementation of `discNat` in Glasgow Haskell, utilizing `unsafePerformIO` to trick Haskell into assigning a purely functional type to basic equivalence discriminators returned by `discNat`. It is given in Figure 10. It corresponds to Cai and Paige's *basic bag discrimination* algorithm (Cai and Paige 1995, Section 2.2), but without requiring uninitialized arrays, as originally described by Paige and Tarjan (1987). As we shall see in Section 13, it has, in contrast to `discNatST` or an implementation based on purely functional arrays,[10] run-time performance competitive with the best comparison-based sorting methods available in Haskell. As

[10] A Haskell implementation using the Data.Array library turns out to be 2 orders of magnitude slower (!). To avoid tempting anybody into running it, it is not reproduced here.

```
discColl update e xss = disc (listE (NatE (length keyNumBlocks - 1))) yss
     where
     (kss, vs)           = unzip xss
     elemKeyNumAssocs    = groupNum kss
     keyNumBlocks        = disc e elemKeyNumAssocs
     keyNumElemNumAssocs = groupNum keyNumBlocks
     sigs                = bdiscNat (length kss) update keyNumElemNumAssocs
     yss                 = zip sigs vs
```

Figure 11: Bag and set equivalence discrimination

noted, care must be exercised, however, since the functions returned by `discNat` are neither thread-safe nor reentrant.

In `disc` we make do with a *single* basic equivalence discriminator, requiring only one global bucket table shared amongst all equivalence discriminators:

```
discNat16 :: Disc Int
discNat16 = discNat 65535
```

When discriminating integers we make a case distinction:

```
disc (NatE n) xs        =
   if n < 65536 then discNat16 xs else disc eqInt32 xs
```

For `Int`-keys whose upper (most significant) 16 bits are all 0s—keys in the range $\{0, \ldots, 65535\}$—we invoke `discNat16` directly. For keys with a non-0 bit in the upper half of a 32-bit word, the call to `disc eqInt32` results in first calling `discNat16` on the upper 16-bit word halves to partition the lower 16-bit word halves, which are then processed by `discNat16` again. This results in each 32-bit key being traversed at most twice.

10.2 Bag and set equivalence discrimination

In Section 7.4 we have seen how to perform bag order discrimination, which treats all permutations of a list as equivalent, by sorting the lists first and then performing lexicographic list order discrimination.

For `BagE e` it would seem we have a problem: How to implement `disc (BagE e)` if there is no order to sort the lists with, only an *equivalence* relation? The key insight, due to Paige (1991, 1994), is that we do not need to sort the lists making up the keys according to a particular ordering relation, but that *any* ordering relation on the actually occurring key elements will do. Paige called sorting multiple lists according to a common ad-hoc order *weak sorting*.

We refine Paige's idea by not returning the key elements themselves, but returning their ordinal numbers in the ad-hoc ordering. This is done by using `disc` instead of `sdisc`. The clause

```
disc (BagE e) xs        = discColl updateBag e xs
```

for processing bag and set equivalence in Figure 8 employs the auxiliary function `discColl`, which is presented in Figure 11. Its only difference to `sdiscColl` in Figure 5 is that it calls `disc` instead of `sdisc`. Consider in particular

```
keyNumBlocks          = disc e elemKeyNumAssocs
keyNumElemNumAssocs   = groupNum keyNumBlocks
```

Here, groups of key indices containing e-equivalent key elements are returned in *some* order, and the subsequent group numbering associates a particular number with each key element occurring in any key. The call

```
sigs                  = bdiscNat (length kss) update keyNumElemNumAssocs
```

returns sorted groups of such key element numbers, which are then used in the call

```
disc (listE (NatE (length keyNumBlocks - 1))) yss
```

to perform list equivalence discrimination.

Example 10.1 For illustration of bag equivalence discrimination let us trace the execution of `disc (BagE eqChar8)` on the input

$$\texttt{xss} = [(\texttt{"bac"}, 1), (\texttt{"ac"}, 2), (\texttt{"cab"}, 3), (\texttt{""}, 4), (\texttt{"ac"}, 5)]$$

from Example 7.4 and Example 7.1, where we have used it for bag order discrimination and list order discrimination, respectively.

The initial steps are the same as for bag order discrimination, resulting in the binding

```
elemKeyNumAssocs = [('b', 0), ('a', 0), ('c', 0),
                    ('a', 1), ('c', 1),
                    ('c', 2), ('a', 2), ('b', 2),
                    ('a', 4), ('c', 4) ].
```

Now, we discriminate these pairs according to the key element equivalence `eqChar8`:

```
keyNumBlocks          = disc eqChar8 elemKeyNumAssocs,
```

which results in

```
keyNumBlocks = [ [0, 2], [0, 1, 2, 4], [0, 1, 2, 4] ]
```

in our example. The groups of key indices are not listed in alphabetic order, but in occurrence order: Since the first occurrence of `'b'` occurs before the first occurrence of `'a'`, which in turn occurs before the first occurrence of `'c'`, the group of indices `[0, 2]` of the keys containing `'b'` occur first, `[0, 1, 2, 4]` containing `'a'` next, and, finally, again `[0, 1, 2, 4]` containing `'c'`, last.

Next we convert `keyNumBlocks` into its group number representation:

```
keyNumElemNumAssocs = groupNum keyNumBlocks,
```

which results in the binding

```
keyNumElemAssocs = [ (0, 0), (2, 0),
                     (0, 1), (1, 1), (2, 1), (4, 1),
                     (0, 2), (1, 2), (2, 2), (4, 2) ].
```

We now discriminate `keyNumElemAssocs`:

```
sigs          = bdiscNat 5 updateBag keyNumElemNumAssocs
```

The resulting signatures are

```
sigs = [ [0, 1, 2], [1, 2], [0, 1, 2], [], [1, 2] ].
```

Observe that they represent the lexicographically ordered keys

```
["bac", "ac", "bac", "", "ac"]
```

under the ad-hoc ordering `'b'` $<$ `'a'` $<$ `'c'`.

Finally, zipping `sigs` with the value components `vs` from the original `xss` gives

```
yss = [([0, 1, 2], 1), ([1, 2], 2), ([0, 1, 2], 3), ([], 4), ([1, 2], 5) ].
```

Applying the list order discriminator

```
disc (listE (NatE (length keyNumBlocks - 1))) yss
```

yields the final output

```
[ [4], [1, 3], [2, 5] ],
```

which, though computed differently, is the same as for bag order discrimination. □

Discrimination for set equivalence is done similar to bag equivalence.

10.3 Correctness and complexity

Theorem 10.2 *Let* e `:: Equiv T`. *Then* `disc e` *is a stable discriminator for* $\mathcal{E}[\![e]\!]$ *over* T.

PROOF (Sketch) Analogous to the proof of Theorem 7.5: The domain-theoretic construction of $\mathcal{E}[\![e]\!]$ gives rise to the notion of rank, which can then be used to prove that the theorem is true for all inputs with keys of finite rank. (Note that the definition of discriminator requires a discriminator only to be defined on keys of finite rank.) □

Analogous to definitions employed in Theorem 8.10, `disc e` executes in linear time for a large class of equivalence representations.

Theorem 10.3 *Let* $T = \mu\texttt{t1}.T'$ *with* $f : T \to T'[T/\texttt{t1}]$ *the unfold-function from* T*. Let* e `:: Equiv T` *and finite* e' `:: Order T'` *such that*

$$e = \texttt{MapE}\ f\ (e'[e/\texttt{e1}])$$

where e, e' *are equivalence representations over* T *and* T'*, respectively,* `t1` *is a formal type variable and* `e1 :: Order t1` *a formal equivalence variable.*

Then `disc e` *executes in linear time.*

PROOF Analogous to the proof of Theorem 8.10. □

11 Representation independence

In the introduction we have motivated the importance of representation independence for discriminators without, however, formalizing it. In this section we define precisely two levels of representation independence, *partial and full abstraction*; point out that `sdisc` is fully abstract for ordering relations; analyze the representation independence properties of `disc`; and show how to make it fully abstract.

11.1 Partial and full abstraction

Definition 11.1 [Key equivalence] Let P be a binary relation. Lists $\vec{x}$ and $\vec{y}$ are *key equivalent under* P if $\vec{x}\,(P \times id)\,\vec{y}$. □

Definition 11.2 [Partially abstract discriminator] A discriminator $\mathcal{D}$ for equivalence relation E is *partially abstract* if $\mathcal{D}(\vec{x}) = \mathcal{D}(\vec{y})$ whenever $\vec{x}$ and $\vec{y}$ are key equivalent under E. □

Combining this property with the parametricity property of Definition 4.2, a partially abstract discriminator for equivalence relation E satisfies, for all Q, $\mathcal{D}(\vec{x})\,Q^{**}\,\mathcal{D}(\vec{y})$ whenever $\vec{x}\,(E \times Q)^*\vec{y}$.

Partial abstraction protects against the effect of replacing a key by an equivalent one becoming observable in the result of a discriminator. But what if we replace *all* keys in the input to a discriminator by completely different ones, but such that the pairwise equivalences are the same as before? Consider again the case of reference discrimination in Section 1, where references are represented by raw machine addresses. Since the raw machine addresses may be different between multiple runs of the same program and furthermore be subject to changes due to copying garbage collection, the result of discrimination with references as keys should only depend on which pairwise equalities hold on the keys in the input and *nothing else*.

Definition 11.3 [P-correspondence] Let P be a binary relation. We say that lists $\vec{x} = [(k_1, v_1), \ldots, (k_m, v_m)]$ and $\vec{l} = [(l_1, w_1), \ldots, (l_n, w_n)]$ are *P-correspondent* and write $\vec{x} \approx_P \vec{y}$ if $m = n$ and for all $i, j \in \{1 \ldots n\}$ we have $v_i = w_i$ and $k_i \, P \, k_j \Leftrightarrow l_i \, P \, l_j$. □

Definition 11.4 [Fully abstract discriminator] A discriminator $\mathcal{D}$ for equivalence relation E is *fully abstract* if it makes P-correspondent inputs indistinguishable: For all $\vec{x}, \vec{y}$, if $\vec{x} \approx_P \vec{y}$ then $\mathcal{D}(\vec{x}) = \mathcal{D}(\vec{y})$.

Likewise, an order discriminator for ordering relation R is *fully abstract* if it makes R-correspondent inputs indistinguishable. □

If E is an equivalence relation, it is easy to see that E-correspondence implies key-equivalence under E:

Proposition 11.5 *If $\vec{x} \, (E^* \times id) \vec{y}$ then $\vec{x} \approx_E \vec{y}$.*

The converse does not hold: $[(4, "A"), (4, "B")] \approx_= [(7, "A"), (7, "B")]$ but, obviously, $[(4, "A"), (4, "B")] \neq [(7, "A"), (7, "B")]$.

Proposition 11.6 *Let $\mathcal{D}$ be a discriminator for E. If $\mathcal{D}$ is fully abstract then it is partially abstract.*

Full abstraction is thus a stronger property than partial abstraction, which explains our choice of terminology.

11.2 Full abstraction of generic order discrimination

Proposition 11.7 `sdisc` *r is a fully abstract discriminator for $\mathcal{O}[\![r]\!]$.*

PROOF This follows from `sdisc` being a stable order discriminator. □

Observe that, even though `sdisc` r is fully abstract as an order discriminator for $\mathcal{O}[\![r]\!]$, it is *not* fully abstract as a discriminator for the equivalence relation $\equiv_{\mathcal{O}[\![r]\!]}$. This is for the simple reason that it always returns its groups in ascending order, making the key ordering observable. Full abstraction for $\equiv_{\mathcal{O}[\![r]\!]}$ would require it to *ignore* the order, which is anathema to the discriminator being an order discriminator to start with.

Example 11.8 Consider the discriminator `sdisc ordNat8` applied to $[(5, "\texttt{foo}"), (8, "\texttt{bar}"), (5, "\texttt{baz}")]$. It returns $[["\texttt{foo}", "\texttt{baz}"], ["\texttt{bar}"]]$, and applied to $[(6, "\texttt{foo}"), (1, "\texttt{bar}"), (6, "\texttt{baz}")]$ it returns $[["\texttt{bar}"], ["\texttt{foo}", "\texttt{baz}"]]$.

Note that $[(5, "\texttt{foo}"), (8, "\texttt{bar}"), (5, "\texttt{baz}")]$ and $[(6, "\texttt{foo}"), (1, "\texttt{bar}"), (6, "\texttt{baz}")]$ are $=$-correspondent, where $=$ denotes equality on unsigned 8-bit integers. By Definition 11.4, a discriminator that is fully abstract under $=$ must return the *same* result for these two inputs. Clearly `sdisc ordNat8` does not. □

11.3 Representation independence properties of generic equivalence discrimination

As discussed in the introduction, our intention is for a discriminator for an equivalence relation to be representation-independent: The result should only depend on the pairwise equivalences that hold on the key components of an input, not the key values themselves in any other way. In other words, it should behave as if it were programmed using a binary equivalence test only, but it should execute a lot faster. Let us consider the equivalence constructors, starting with integer segment equality.

Theorem 11.9 *The basic equivalence discriminator* `discNat n` *from Section 10.1 is fully abstract under equality on* $\{0, \ldots n\}$.

PROOF The algorithm builds a list of unique keys in the order of their first occurrence in the input and then traverses the list to output the associated groups of values. For correspondent inputs there is a one-to-one mapping between keys in one input and in the other input such that the respective unique key lists are, elementwise, in that relation. Consequently, outputting the associated values in order of the key lists yields the same groups of values in both cases. □

This is a best-case scenario: the basic equivalence discriminator is not only *efficient* because it ignores the key order, but precisely because of that it is also *fully abstract*!

Unfortunately, the equivalence discriminators for sum and product equivalences only preserve partial abstraction, and for bag and set equivalences we do not even get partially abstract discriminators.

Proposition 11.10 `disc` *is partially abstract for equivalences not containing* `BagE` *or* `SetE`.

As the following example shows, this proposition unfortunately does not extend to bag and set equivalences.

Example 11.11 Since `"ab"` and `"ba"` are `BagE eqChar8`-equivalent, `[("ab", 1), ("a", 2), ("b", 3)]` and `[("ba", 1), ("a", 2), ("b", 3)]` are key-equivalent under `BagE eqChar8`-equivalence. We have that `disc (BagE eqChar8) [("ab", 1), ("a", 2), ("b", 3)]` evaluates to `[[2],[1],[3]]`, but `disc (BagE eqChar8) [("ba", 1), ("a", 2), ("b", 3)]` evaluates to `[[3],[1],[2]]`. □

If fully abstract equivalence discrimination is required, we can accomplish it by sorting the value groups returned by `disc` according to the position of first occurrence of the first value of an output group in the input. This can be done as follows:

```
edisc' :: Equiv k → Disc k
edisc' e xs = map (map snd)
                  (dsort (ListL (ProdL (NatO (length xs)) TrivO))
                         (disc e xs'))
     where xs' = map relabel (zip xs ([0..] :: [Int]))
           relabel ((k, v), pos) = (k, (pos, v))

edisc :: Equiv k → Disc k
edisc e xs | reqPostProc e = edisc' e xs
edisc e xs | otherwise = disc e xs
   where reqPostProc :: Equiv t → Bool
                reqPostProc (NatE _) = False
                reqPostProc TrivE = False
                reqPostProc (SumE _ _) = True
                reqPostProc (ProdE _ _) = True
                reqPostProc (MapE _ e) = reqPostProc e
                reqPostProc (ListE _) = True
                reqPostProc (BagE _) = True
                reqPostProc (SetE _) = True
```

Figure 12: Fully abstract equivalence discriminators `edisc'` and `edisc`.

1. Label input pairs with their input position.
2. Perform equivalence discrimination using `disc`.
3. Sort groups of values returned in Step 2 by their position labels: List the group with a value occurring before the values of another group before that group.
4. Remove labels.

The sorting step can done by applying the generic sorting function `dsort` (defined in the following section) to a suitable order representation. This illustrates the method of solving a sorting or partitioning problem by finding the "right" ordering relation, respectively equivalence relation. It is captured in the code of `edisc'` presented in Figure 12. Recall that `disc` is stable, which ensures that the position label of the first value in a group is the left-most position of any value in that group. Furthermore, computationally only the first element in each group is inspected by `dsort`, without processing the remaining elements.

For some equivalence representations the sorting step is not necessary. The function `edisc` in Figure 12 first checks the equivalence representation passed to it and only performs the more complex label-discriminate-sort-unlabel steps if it contains an order constructor that does not preserve full abstraction.

```
spart :: Order t → [t] → [[t]]
spart r xs = sdisc r [ (x, x) | x ← xs ]

sort :: Order t → [t] → [t]
sort r xs = [ y | ys ← spart r xs, y ← ys ]

usort :: Order t → [t] → [t]
usort r xs = [ head ys | ys ← spart r xs ]
```

Figure 13: Generic discriminator-based partitioning, sorting and unique-sorting

Theorem 11.12 *Both* `edisc'` *e and* `edisc` *e are fully abstract equivalence discriminators for* $\mathcal{E}[\![e]\!]$.

The position numbering technique is a generally useful instrumentation technique for representing positional order as an ordering relation. It can be used to force a sorting algorithm to produce stable results and to ensure that query results are eventually produced in the semantically specified order despite using intermediate operations that treat them as multisets (Grust et al. 2004).

12 Applications

We present a few applications of order and equivalence discrimination intended to illustrate some of the expressive power of order and equivalence representations and the asymptotic efficiency achieved by generic discrimination.

12.1 Sorting and partitioning by discrimination

Generic sorting and partitioning functions can be straightforwardly defined from generic discriminators.

A list of keys can be partitioned in ascending order by associating each key with itself and then performing an order discrimination:

```
spart :: Order t → [t] → [[t]]
spart r xs = sdisc r [ (x, x) | x ← xs ]
```

By flattening the result of `spart` we obtaining the *discriminator-based* generic sorting function

```
dsort :: Order t → [t] → [t]
dsort r xs = [ y | ys ← spart r xs, y ← ys ].
```

Since `sdisc` produces stable order discriminators, `dsort`, likewise, produces a stable sorting function for each order representation.

Choosing the first element in each group output by `spart`, lets us define a *unique-sorting* function:

```
dusort :: Order t → [t] → [t]
dusort r xs = [ head ys | ys ← spart r xs ]
```

It sorts its input, but retains only one element among equivalent keys. In particular, it can be used to efficiently eliminate duplicates in lists of elements of ordered types. Choosing the first element in each group combined with stability of `sdisc` guarantees that the output of `dusort` contains the first-occurring representative of each equivalence class of input keys. It can be used to eliminate duplicates and put the elements into a canonical order.

The function `part`

```
part :: Equiv t → [t] → [[t]]
part e xs = disc e [ (x, x) | x ← xs ]
```

partitions its input according to the equivalence representation passed to it. The function `reps`

```
reps :: Equiv t → [t] → [t]
reps e xs = [ head ys | ys ← part e xs ]
```

is analogous to `dusort`, but for equivalence representations: it selects a single representative from each equivalence class.

As alternatives, we can use `edisc` instead of `disc` in the definitions of `part` and `reps`:

```
epart :: Equiv t → [t] → [[t]]
epart e xs = edisc e [ (x, x) | x ← xs ]

ereps :: Equiv t → [t] → [t]
ereps e xs = [ head ys | ys ← epart e xs ]
```

The full abstraction and stability properties of `edisc` guarantee that `epart` returns partitions in order of first occurrence (of some element of an equivalence class) in the input; and `ereps` lists the first-occurring representative of each equivalence class. Functions `reps` and `ereps` are analogous to Haskell's `nubBy`, which eliminates E-duplicates from an input list when passed an equality test for E, but `reps` and `ereps` do so asymptotically faster, avoiding the inherent quadratic complexity of `nubBy` due to Proposition 1.1. The performance difference is dramatic in practice. For example, using Glasgow Haskell[11] the call

```
length (nubBy (λ x y → x + 15 == y + 15) [1..n])
```

has super-second performance already for $n \approx 1500$. The corresponding call

```
length (reps (MapE (+ 15) eqInt32) [1..n])
```

[11] See Section 13 for more information on experimental set-up.

still displays sub-second performance for $n \approx 700000$. Even `nub`, when applied to integers, which in Glasgow Haskell runs about 100 times faster than when given a user-defined equality test such as the one above, is dramatically outperformed by `reps` and `ereps`. For example, evaluation of

```
length (reps eqInt32 [1..1000000])
```

takes approximately 1.5 seconds, whereas the corresponding evaluation

```
length (nub [1..1000000])
```

takes about 1.5 hours (!).

12.2 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 indices such that each group represents all the occurrences of the same word in the text. This is accomplished by the following function:

```
occsE :: [(String, Int)] → [[Int]]
occsE = disc eqString8
```

Each group of indices returned points to the same word in the original text. If we wish to return the group in the lexicographic order of the words they index we use `sdisc`:

```
occsO :: [(String, Int)] → [[Int]]
occsO = sdisc ordString8
```

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 equivalence, respectively order representation, correspondingly:

```
ordString8Ins :: Order String
ordString8Ins = listL (MapO toUpper ordChar8)

occsCaseInsE :: [(String, Int)] → [[Int]]
occsCaseInsE = disc (equiv ordString8Ins)

occsCaseInsO :: [(String, Int)] → [[Int]]
occsCaseInsO = sdisc ordString8Ins
```

Here, `toUpper` is function that maps lower-case characters to their upper-case counterparts and acts as the identity on all other characters. We could also use `toLower` instead of `toUpper`, which illustrates that the same order may have multiple representations. The function `equiv` produces a representation of the largest equivalence contained in the ordering denoted by its input. See Section 14.1 for its definition.

12.3 Anagram classes

A classical problem treated by 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:

```
anagrams :: [String] → [[String]]
anagrams = part (BagE eqChar8)
```

This is arguably the shortest solution to Bentley's problem, and it even improves his solution asymptotically: it runs in $O(N)$ time instead of $\Theta(N \log N)$.

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

```
anagramsCaseIns :: [String] → [[String]]
anagramsCaseIns = part (BagE (MapE toUpper eqChar8))
```

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

```
bsbE :: Equiv [[Int]]
bsbE = BagE (SetE eqNat8)
```

Discrimination and partitioning functions are then definable by applying `disc` and `part`, respectively, to `bsbE`.

12.4 Lexicographic sorting

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

```
lexUp = dsort ordString8
lexDown = dsort (Inv ordString8)
lexUpCaseIns = dsort (ListL (MapO toUpper ordChar8))
```

The elements need not be fixed-sized. The corresponding functions for lexicographic sorting of lists of strings are

```
lexUp2 = dsort (ListL ordString8)
lexDown2 = dsort (Inv (ListL ordString8))
lexUpCaseIns2 = dsort (ListL (listL (MapO toUpper ordChar8)))
```

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 Bucketsort-based discriminator `sdiscNat` n. Each call to `sdiscNat` n with a list of m key-value pairs traverses an entire bucket table of size n. So traversal time is $O(n + m)$, which means n dominates for small values of m.

If the output does not need to be alphabetically sorted, traversal time can be made independent of the array size by employing the basic bag discriminator of Figure 10. 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 has been 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:

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

We can see that `sdisc r2` is called for each group `ys` output by the first discrimination step. If `r2` is `NatO n`, the repeated calls of `disc r2` are calls to the bucket sorting based discriminator `sdiscNat n`. 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. It is possible to generalize Forward Radixsort (Andersson and Nilsson 1994, 1998), a left-to-right (most significant digit first, MSD) Radixsort that visits only the minimum distinguishing prefixes *and* avoids sparse bucket table traversals. The idea 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 produces the same groups as in the code above.

Formally, this can be specified as follows:

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

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 strict depth-first processing of groups to full breadth-first processing, which has bad locality. To wit, when using basic equivalence 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 believe that concatenating not *all* groups `ys` returned by `disc r1` in the defining clause for `disc (Pair r1 r2)`, but *just sufficiently many* to fill the bucket table to "pay" for its traversal may lead to a good algorithm that retains the advantages of MSD radix sorting without suffering the cost of near-empty bucket table traversals. Even for the special case of string sorting, this does not seem to have been explored yet, however: Forward Radixsort uses pure breadth-first processing, and other MSD-Radixsort implementations are based on Adaptive Radixsort (Andersson and Nilsson 1998; Maus 2002; Al-Badarneh and El-Aker 2004).

12.5 Type isomorphism

Consider finite type expressions built from type constructors $\times$ (product) and other constructors such as $\rightarrow$ (function type) and *Bool* (Boolean type). We say two type expressions are *A-isomorphic* if one can be transformed into the other using equational rewriting and associativity of the product constructor: $(T_1 \times T_2) \times T_3 = T_1 \times (T_2 \times T_3)$ for all T_1, T_2, T_3. The *A-isomorphism problem* for nonrecursive types is the problem of partitioning a set of type expressions into A-isomorphic equivalence classes.

The problem can be solved as follows. We define a data type for 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 transformed 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 a product type. This corresponds to exploiting the associativity property of $\times$. We can use the following data type for representing the transformed type expressions:

```
data TypeExp2 = TCons2 String [TypeExp2]
              | Prod2 [TypeExp2]
```

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

```
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
```

Transformed type expressions are isomorphic if they are structurally equal, which is denoted by the following equivalence representation:

```
prod2 :: Order TypeExp2
prod2 = MapE unTypeExp2
        (SumE (ProdE eqString8 (ListE prod2)) (ListE prod2))
```

where

```
unTypeExp2 (TCons2 v cts) = Left (v, cts)
unTypeExp2 (Prod2 cts) = Right cts
```

is the unfold direction of the isomorphism between `TypeExp2` and `Either (String, [TypeExp2]) [TypeExp2]`.

The complete solution to the type isomorphism problem with an associative type constructor is then

```
typeIsoA :: [TypeExp] → [[TypeExp]]
typeIsoA = part (MapE trans prod2)
```

It is easy to see that `trans` executes in linear time on *unshared* type expressions, and by Theorem 8.10 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 *AC-isomorphism* where the product constructor is both associative and commutative: $T_1 \times T_2 = T_2 \times T_1$ for all T_1, T_2. Application of `trans` handles associativity as before, and commutativity can be captured by the equivalence denoted by

```
prod3 :: Order TypeExp2
prod3 = MapE unTypeExp2
        (SumE (ProdE eqString8 (ListE prod2)) (BagE prod3))
```

The only change to `prod2` is the use of `BagE prod3` instead of `ListL prod2`.

The complete solution to the type isomorphism problem with an associative-commutative type constructor is thus

```
typeIsoAC :: [TypeExp] → [[TypeExp]]
typeIsoAC = part (MapE trans prod3)
```

By Theorem 10.3 `typeIsoAC` executes in worst-case linear 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* bag discrimination for trees with weak sorting (Paige 1991). For pairs of types this has also been proved separately (Zibin et al. 2003; Gil and Zibin 2005), where basic bag discrimination techniques due to Cai and Paige (1991, 1995) have been rediscovered.

The above shows that the same result is an instance of our master theorem for linear time discriminability for equivalences. In particular, bottom-up multi-set discrimination is not required, as previously claimed (Henglein 2008). Our bag and set equivalence discrimination techniques of Section 10.2 are sufficient.

The type isomorphism problem with an associative-commutative product type constructor is a special case of the term equality (isomorphism) problem with free, associative, associative-commutative and associative-commutative-idempotent operators. By generalizing `trans` to work on multiple associative operators and using `BagE` for commutative operators and `SetE` for commutative-idempotent operators, the solution above can be generalized to a linear-time solution for the general term equality problem.[12]

12.6 Discrimination-based joins

Relational queries are conveniently represented by list comprehensions (Trinder and Wadler 1988). For example,

```
[(dep, acct) | dep ← depositors, acct ← accounts,
               depNum dep == acctNum account ]
```

computes the list of depositor/account-pairs with the same account number.

The problem is that a naive execution of the query is inadvisable since it explicitly iterates through the Cartesian product of depositors and accounts before filtering most of them out again.[13] For this reason, database systems employ efficient *join* algorithms for performing the filtering without iterating over all the elements of the Cartesian product explicitly.

We show how to implement an efficient *generic* join algorithm for a large class of equivalence relations by using the generic discriminator `disc` in Figure 8.

To start with, let us define types for the entities of relational algebra: sets, projections and predicates.

```
data Set a = Set [a]
data Proj a b = Proj (a → b)
data Pred a = Pred (a → Bool)
```

Note that these definitions generalize relational algebra: sets may be of any type, not just records of primitive types; we allow arbitrary functions, not only projections on records; predicates may be specified by any Boolean function, not just equality and inequality predicates involving projections.

The core relational algebra operators `select, project, prod` then correspond to `filter, map` and explicit Cartesian product construction:

[12] It should be emphasized that it is linear in the *tree size* of the input terms. The linear time bound does not apply to the *graph size* of terms represented as acyclic graphs.

[13] It is even worse if the Cartesian product is materialized. Haskell's lazy evaluation avoids this, however.

```
select :: Pred a → Set a → Set a
select (Pred c) (Set xs) = Set (filter c xs)

project :: Proj a b → Set a → Set b
project (Proj f) (Set xs) = Set (map f xs)

prod :: Set a → Set b → Set (a, b)
prod (Set xs) (Set ys) = Set [(x, y) | x ← xs, y ← ys]
```

Using the above operators our example can be written as

```
select (Proj λ (dep, acct) → depNum dep == acctNum account)
       (prod depositors accounts)
```

We can add a generic (equi)join operation with the following type:

```
join :: Proj a k → Equiv k → Proj b k → Set a → Set b → Set (a, b)
```

It can naively be implemented as follows:

```
join (Proj f1) e (Proj f2) s1 s2 =
   select (Pred (λ (x, y) → eq e (f1 x) (f2 y))) (prod s1 s2)
```

Using `join` our example query can now be formulated as follows:

```
join (Proj depNum) eqNat16 (Proj acctNum) depositors accounts
```

if all account numbers are in the range $[0 \ldots 65535]$. (If account numbers can be arbitrary 32-bit integers, we simply replace `eqNat16` above by `eqInt32`.) Nothing is gained, however, without a more efficient implementation of `join`: the time complexity is still $\Theta(mn)$ if m, n are the number of records in `depositors`, respectively `accounts`.

The key idea in improving performance is that the result of `join (Proj f1) e (Proj f2) s1 s2` consists of the union of the Cartesian products of records `x, y` from `s1, s2`, respectively, such that `f1 x` and `f2 y` are `e`-equivalent.

Usually hashing or sorting, restricted to equality on atomic types, are used in efficient join-algorithms in a database setting. We show how to do this using generic equivalence discrimination for arbitrary denotable equivalence relations, including for complex types and references, which have neither an ordering relation nor a hash function.[14]

The following describes the steps:

1. Form the lists $[(f_1(x), inl\, x) \mid x \in s_1]$ and $[(f_2(y), inr\, y) \mid y \in s_2]$ and concatenate them.

2. Apply `disc e` to this list.

[14] We do not discuss the requirements of I/O efficiency for data stored on disk here, but appeal to the scenario where the input data are stored or produced in main memory.

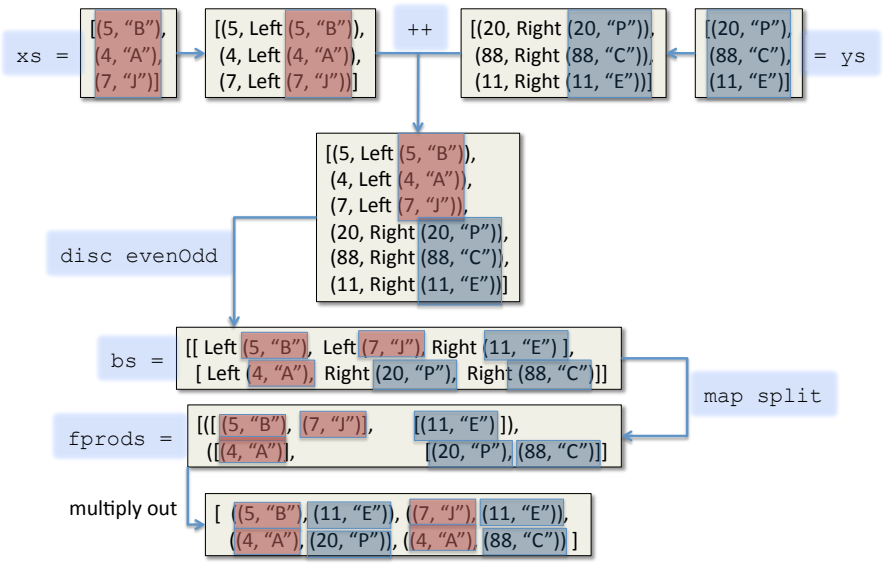

Figure 14: Example: Execution of discrimination-based join

3. Each group in the result of the discriminator consists of records from s_1 and s_2. Compute the Cartesian product of the s_1-records with the s_2-records for each group, and finally concatenate the Cartesian products for each group.

This can be coded as follows:

```
join :: Proj a k → Equiv k → Proj b k → Set a → Set b → Set (a, b)
join (Proj f1) e (Proj f2) (Set xs) (Set ys) =
     Set [ (x, y) | (xs, ys) ← fprods, x ← xs, y ← ys ]
   where bs = disc e ([(f1 x, Left x) | x ← xs] ++
                      [(f2 y, Right y) | y ← ys])
         fprods = map split bs
```

Figure 14 illustrates the evaluation of

```
join (Proj fst) (Equiv evenOdd) (Proj fst)
          (Set [(5, "B"), (4, "A"), (7, "J")])
          (Set [(20, "P"), (88, "C"), (11, "E")])
```

Recall that `evenOdd = MapE (‘mod‘ 2) (NatE 1)` denotes the equivalence E_{eo} of Example 4.4.

With this implementation of `join` the query

```
join (Proj depNum) eqNat16 (Proj acctNum) depositors accounts
```

executes in time linear in the size of its input and output.

Note that this *discriminatory-join* algorithm admits complex element types and equivalence relations on them such as bag-equivalence, which is not supported in ordinary relational algebra or MapReduce-frameworks, and it still works in worst-case linear time.

The tagging of records before submitting them to a discriminator and the subsequent separation can be avoided by employing a *binary* discriminator `disc2`, which can be defined generically, completely analogous to the definition of `disc`.

Query evaluation can be further improved by using lazy (symbolic) data structures for representing Cartesian products and unions (Henglein 2010; Henglein and Larsen 2010).

13 Performance

We have shown that the generic top-down discriminators `sdisc`, `disc` and `edisc` are representation independent—in the case of `disc` to a limited degree—and asymptotically efficient in the worst case. In this section we take a look at the practical run-time performance of our discriminators and compare them to comparison-based sorting algorithms in Haskell.

Drawing general conclusions about the performance of discrimination from benchmark figures is difficult for a number of obvious reasons: Applying descriptive statistical methods per se allows drawing conclusions only for the particular benchmark suite under scrutiny. Employing inferential statistical methods to extend conclusions to a larger data set requires careful experimental design with random sampling, blind and double blind set-ups and such. Furthermore, the performance measured reflects the amalgam of the algorithm, its particular implementation, the language it is implemented in, the particular compiler, run-time system and machine it is executed on. Haskell employs lazy evaluation, asynchronous garbage collection and a memory model that leaves it to the compiler how to represent and where to allocate data in memory, which makes for convenient high-level programming, but also makes interpretation of performance results tenuous.

Having stated this general disclaimer, we pose the following two hypotheses and set out to support them empirically in this section.

- Equivalence discrimination using `disc` is radically more efficient than discrimination or partitioning using an equivalence test only.
- The time performance of `sdisc` and `disc` is competitive with (and in some cases superior to) standard comparison-based sorting algorithms.

We furthermore believe that generic discrimination is a promising basis for engineering highly run-time efficient code for modern parallel computer architectures, notably GPGPU, multicore and MapReduce-style (Dean and

Ghemawat 2004) distributed compute server architectures. This is not investigated here, however.

The first hypothesis is easy to validate. Proposition 1.1 shows that a partitioning algorithm using only equivalence tests requires a quadratic number of equivalence tests. In Section 12.1 we have seen that even for small data sets (say 100,000 keys) such an algorithm is no longer usable on the current generation of personal computers and that `disc`-based partitioning operates in the subsecond range on such data sets.

To investigate the second hypothesis we perform two experiments on randomly generated inputs. In each case we discriminate inputs whose keys are lists of integers. In the first experiment we discriminate for the standard lexicographic ordering on integer lists. Note that its induced equivalence relation is list equality. In the second experiment we discriminate the same inputs, but for the *Dershowitz-Manna multiset ordering*, respectively bag equivalence.

The Dershowitz-Manna multiset ordering (Dershowitz and Manna 1979), restricted to total preorders (Jouannaud and Lescanne 1982), is denoted by

```
multiset :: Order t → Order [t]
multiset r = MapO (dsort (Inv r)) (ListL r).
```

It is well-founded if and if its element ordering is well-founded, which has applications in proving termination of rewriting systems. The only difference of `multiset r` to `BagO r` is that the former sorts lists in descending order instead of ascending order, before comparing them according to lexicographic list ordering.

The list of keys $[k_1, \ldots, k_i, \ldots]$ in the input to a discriminator is pseudo-randomly generated from the following three parameters:

List length parameter m**:** The length $|k_i|$ of each list k_i making up a key is uniformly randomly chosen from the range $[0 \ldots m-1]$.

Range parameter N**:** The elements of each list k_i are drawn uniformly randomly from the range $[0 \ldots N-1]$.

Total number parameter n**:** Random lists k_i are added to the keys until $(\sum_{k_i} |k_i|) \geq n$.

The input to a discriminator is formed by zipping the keys with $[1 \ldots]$.

A *comparison-parameterized discriminator* employs a comparison-based sorting algorithm: The input is first sorted on the keys, and finally the values associated with runs of equivalent keys are returned. We implement three comparison-parameterized discriminators named `cdisc`, `qdisc` and `mdisc`, based on the following functional versions of sorting algorithms, respectively:

sortBy: The standard Haskell sorting function `sortBy`.

Label	Discriminator
sortBy	cdisc (<=)
qsort	qdisc (<=)
msort	mdisc (<=)
sdisc	sdisc (ListL ordNat8)
disc	disc (ListE eqNat16)
sortBy (bag)	cdisc slte
qsort (bag)	qdisc slte
mdisc (bag)	mdisc slte
sortBy (bag eff)	cdisc (<=) . map sortFst
qsort (bag eff)	qdisc (<=) . map sortFst
msort (bag eff)	mdisc (<=) . map sortFst
sdisc (bag)	sdisc (multiset ordNat8)
disc (bag)	disc (BagE eqNat16)

Figure 15: Discriminators used in performance tests

qsort: Quicksort, with the median of the first, middle and last key in the input being the pivot.

msort: Top-down Mergesort.

Figure 15 shows which discriminators have been tested. The first 5 discriminate integer lists under their standard lexicographic ordering (sortBy, qsort, msort, sdisc), respectively list equality (disc). The remaining discriminators are for the Dershowitz-Manna multiset ordering. The first 3 of these, labeled "bag", are passed a comparison function

```
slte x y = sort x ≤ sort y
```

that first sorts its two argument lists and then performs a standard lexicographic comparison. This causes the sorting step to be applied multiple times on the same key. The following 3, labeled "bag eff", avoid this by sorting each input list exactly once

```
sortFst (x, n) = (sort x, n)
```

and then passing the result to a discriminator for lexicographic ordering.

The test results presented in Figures 16 to 19 have been performed with parameters $N = 256$ and $n = 100000, 200000, \ldots, 1000000$. Figures 16 and 17 show the run times for short keys, which are generated using parameter value $m = 10$. Figure 18 shows the run times for $m = 1000$. Finally, Figure 19 shows them for $m = 10000$. All tests have been performed under Glasgow Haskell, version 6.10.1, on a 2.4 MHz dual-core MacBook Pro 4,1 with 3 MB of level 2 cache, 4 GB of main memory and 800 MHz bus speed, running MacOS X 10.5.8. The run-times are computed as the average of

10 runs using GHC's getCPUTime function. The time measured excludes initial traversal of the input to ensure that it is fully evaluated, but includes traversing the output, which ensures that it is fully evaluated. The tests were compiled using the "-O" flag.[15]

The run times in Figure 16 are given as a function of the minimum distinguishing prefix of the integer lists serving as keys since all the discriminators used for lexicographic ordering/equality only inspect the minimum distinguishing prefix in the input.

Since the multiset ordering/bag equivalence discriminators traverse all elements of each key, the run-times in the other figures are given as a function of the total input size.

Both the input size and minimum distinguishing prefix size are computed from the input as the number of 32-bit words used to store the input in fully boxed representation.

Figure 16 indicates that `sdisc` and `disc` are competitive with comparison-based sorting for lexicographic ordering. The numbers observed are favorable for discrimination, but it should be observed that they exploit that the integer lists contain only small numbers. Executing `disc eqInt32`, which works for all 32-bit integers, adds about 30% to the run time of `disc eqNat16` used in the test since each integer is scanned twice, once for each of its 16-bit halfwords.

The upper chart in Figure 17 shows the costs of calling a comparison-based sorting discriminator with a complex comparison function. The lower chart is a blow-up of the performance of the 5 efficient bag discriminators. Note that the inputs are the same in Figures 16 and 17.

Figures 18 and 19 show the running times for medium-sized (up to 1000 elements) and large (up to 10000 elements) keys, respectively. Here `disc (BagE eqNat16)` behaves comparably to the other discriminators. Its performance is not as favorable as for lexicographic equality, presumably due to the more complex processing involved in performing weak sorting. Indeed running it with `BagE eqInt32` adds about 50% to its run time.

In summary, the tests provide support for our hypothesis that discrimination without any form of program optimization such as specialization, imperative memory management, etc., has acceptable performance and is competitive with straightforwardly coded functional comparison-based sorting algorithms.

14 Discussion

In this section we discuss a number of points related to discrimination.

[15] Full source code of the tests can be retrieved from the author's web page for validation and experimentation.

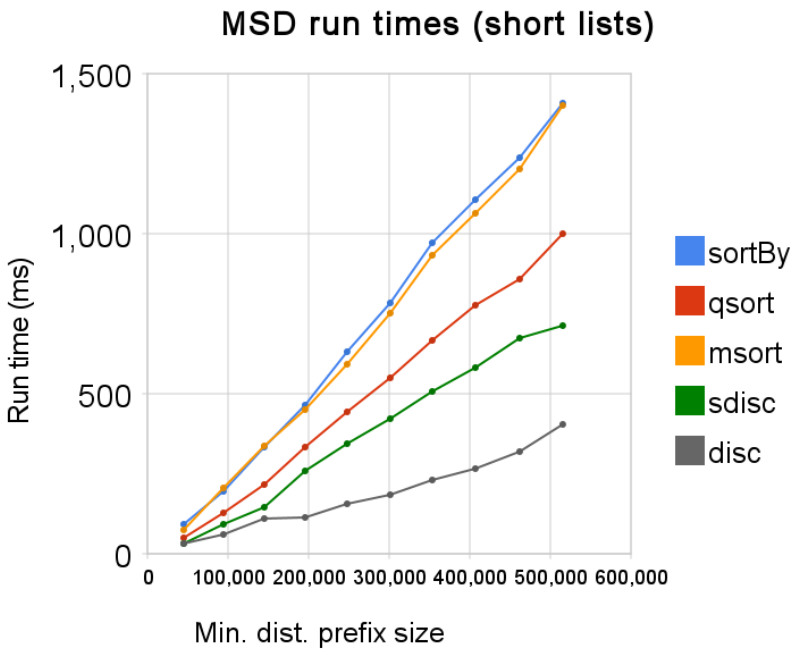

Figure 16: Discriminator execution times for keys made up of short lists of small integers

14.1 Discrimination combinators

Since the generic discriminators `sdisc` and `disc` are defined by structural recursion over order representations, respectively equivalence representations, such expressions can be eliminated by partial evaluation, resulting 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 representations (one per type) to order/equivalence representations (many per type). Figure 20 illustrates the result of doing this for order discriminators. Similarly, we can define ordered partitioning and sorting functions by passing them a discriminator; see Figure 21.

The advantage of the discriminator combinator library in Figure 20 vis a vis the generic discriminator is that it does away with explicit representations of orders and equivalences altogether and lets programmers compose discriminators combinatorially. In particular, the use of GADTs can be avoided altogether if rank-2 polymorphism is available. Also, it incurs no run-time overhead for representation processing.[16]

The disadvantage is that user-definable computation on orders and equivalences is no longer possible. For example, if a user wishes to use order

[16]The generic discriminator `sdisc` appears to execute noticeably *more* efficiently than the combinator library in Glasgow Haskell, however.

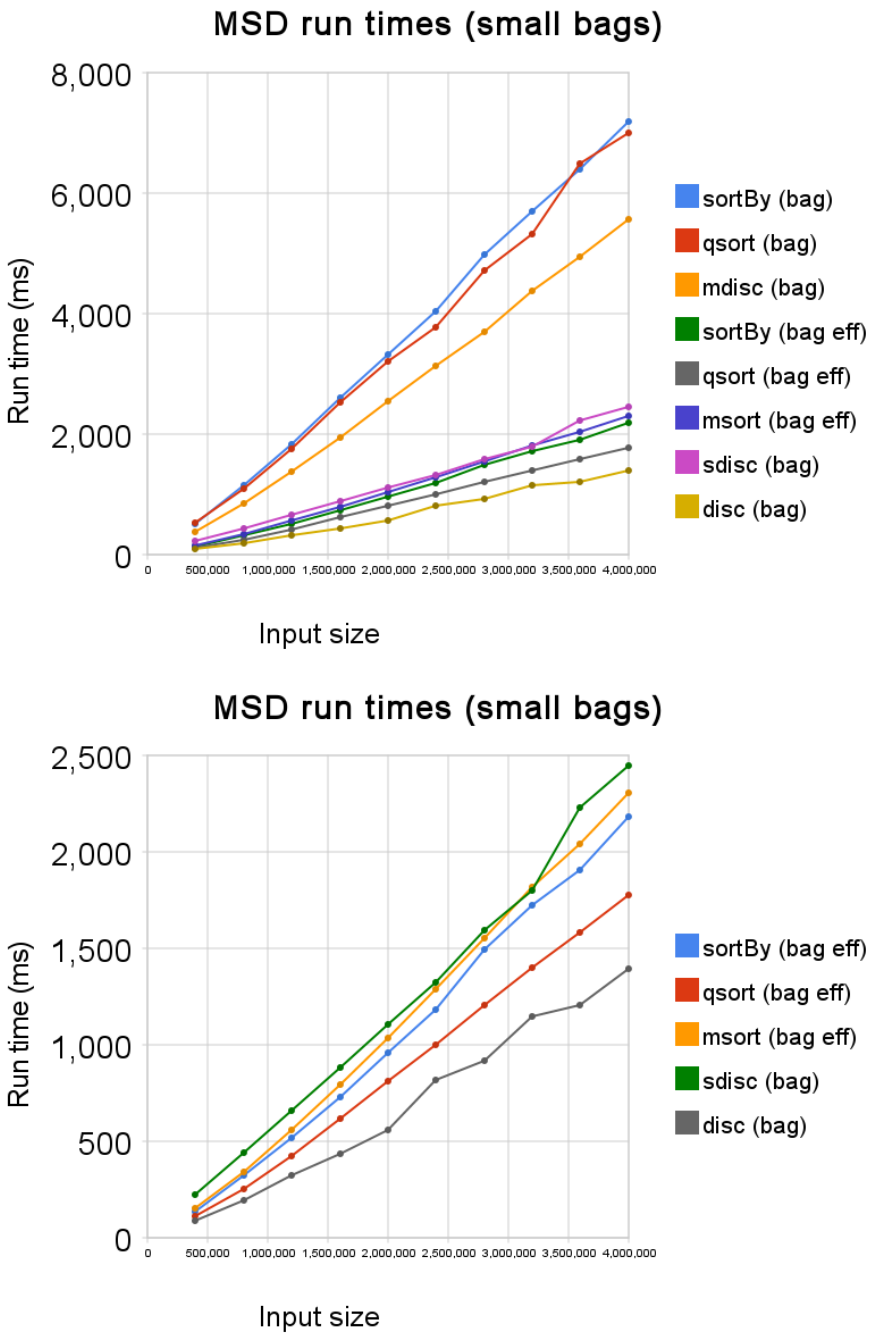

Figure 17: Discriminator execution times for keys made up of small bags of small integers

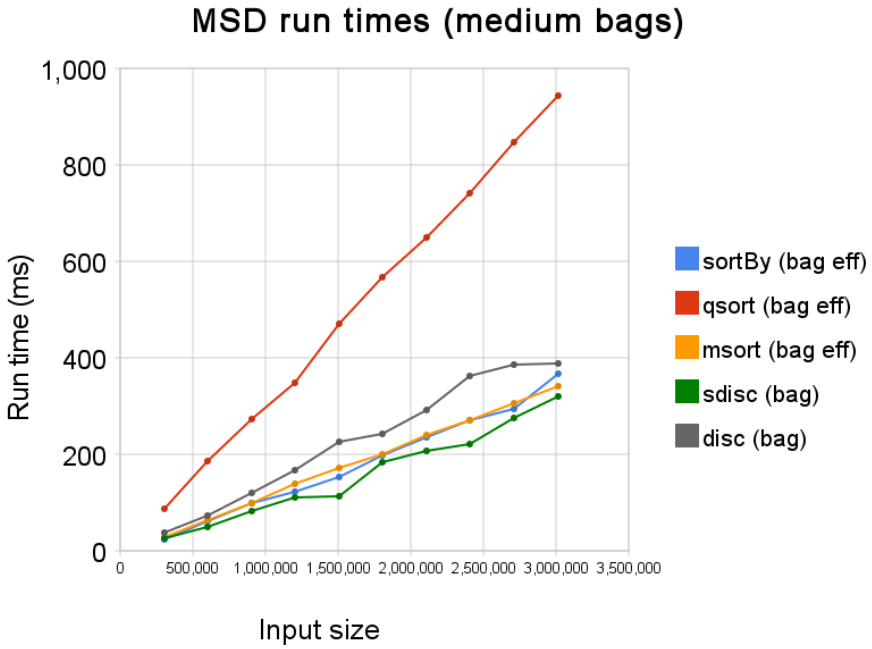

Figure 18: Discriminator execution times for keys made up of medium-sized bags of small integers

representations as input to the equivalence discriminator `disc`, this can be done by providing the function `equiv` in Figure 22, which computes a representation of the equivalence induced by (the ordering relation denoted by) an order representation.

Another example is the function `simplify` in Figure 23, which simplifies an order representation prior to submitting it to `sdisc`. It does not change the denoted order, but, when passed to `sdisc`, may eliminate potentially costly traversals of the input data. Note that variations are possible, which may prove advantageous depending on their use; e.g. simplifying `PairL TrivO TrivO` to `TrivO` and `MapO f TrivO` to `TrivO`.[17] (Recall that f in order representations must be total.)

14.2 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 RAMs, 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

[17] These simplifications have been suggested by one of the referees.

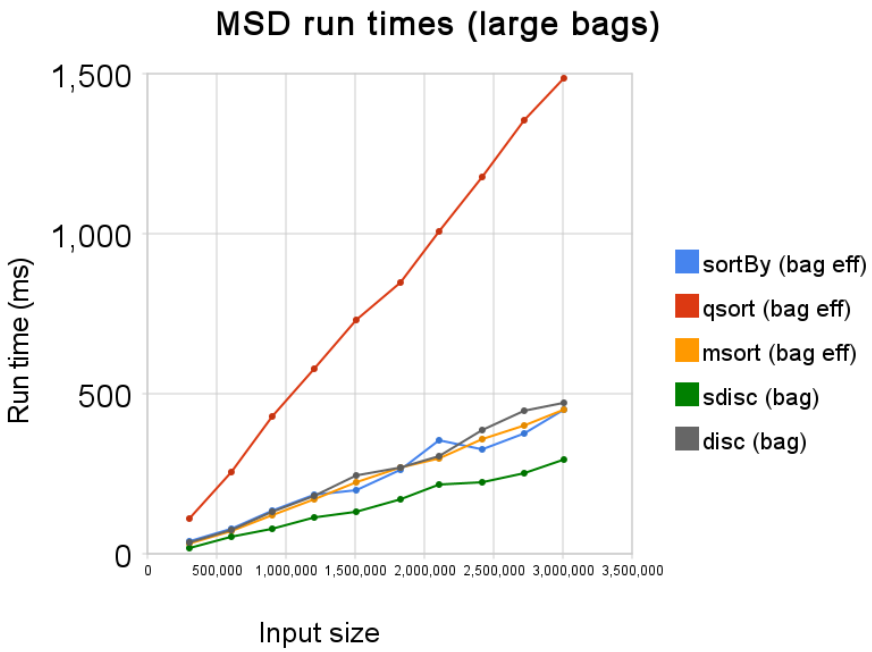

Figure 19: Discriminator execution times for keys made up of large bags of small integers

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 in Section 8 is a random-access machine with *fixed* word width w, say 32 or 64 bits, corresponding to a conventional sequential computer. We treat the word width as a constant factor in asymptotic complexity analysis. In other words, we do not analyze the dependency of the complexity on the word width. Intuitively, this is tantamount to saying that each primitive operation in our model of computation requires time proportional to w: The time to process N bits stored in N/w_1 words of width w_1 is proportional to the time to process them in N/w_2 words of width w_2. This is in contrast to RAM models with word-level parallelism, where primitive operations execute in time *independent* of w; e.g., addition on a 4-bit machine takes the same time as addition on a 1,000,000-bit machine. Such a model emphasizes the benefits of algorithms that manage to exploit the

```
sdiscTriv0 :: Disc k
sdiscTriv0 xs = [[ v | (_, v) ← xs ]]

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

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

sdiscMap0 :: (k1 → k2) → Disc k2 → Disc k1
sdiscMap0 f d xs  = d [ (f k, v) | (k, v) ← xs ]

sdiscListL :: Disc k → Disc [k]
sdiscListL d xs = case nilVals of
                    [] → bs
                    _  → nilVals : bs
   where splitL [] = ([], [])
         splitL ((ks, v) : xs) =
            case ks of
              [] → (v : nilVals, pairVals)
              (k : ks') → (nilVals, (k, (ks', v)) : pairVals)
            where (nilVals, pairVals) = splitL xs
         (nilVals, pairVals) = splitL xs
         bs = [ vs | ys ← d pairVals, vs ← sdiscListL d ys ]

sdiscBag0 d xs = sdiscColl0 updateBag d xs
sdiscSet0 d xs = sdiscColl0 updateSet d xs

sdiscColl0 update d xss = sdiscListL (sdiscNat (length keyNumBlocks)) yss
     where
     (kss, vs)           = unzip xss
     elemKeyNumAssocs    = groupNum kss
     keyNumBlocks        = d elemKeyNumAssocs
     keyNumElemNumAssocs = groupNum keyNumBlocks
     sigs                = bdiscNat (length kss) update keyNumElemNumAssocs
     yss                 = zip sigs vs

sdiscInv :: Disc k → Disc k
sdiscInv d xs       = reverse (d xs)

sdiscChar8 = sdiscMap0 ord (sdiscNat 65535)
sdiscString8 = sdiscListL sdiscChar8
```

Figure 20: Order discriminator combinators

```
spartD :: SDisc t t → [t] → [[t]]
spartD d xs = d [ (x, x) | x ← xs ]

dsortD :: SDisc t t → [t] → [t]
dsortD d xs = [ y | ys ← spartD d xs, y ← ys ]

usortD :: SDisc t t → [t] → [t]
usortD d xs = [ head ys | ys ← spartD d xs ]
```

Figure 21: Discriminator-parameterized partitioning and sorting

```
equiv :: Order t → Equiv t
equiv (NatO n) = NatE n
equiv TrivO = TrivE
equiv (SumL r1 r2) = SumE (equiv r1) (equiv r2)
equiv (ProdL r1 r2) = ProdE (equiv r1) (equiv r2)
equiv (MapO f r) = MapE f (equiv r)
equiv (ListL r) = ListE (equiv r)
equiv (BagO r) = BagE (equiv r)
equiv (SetO r) = SetE (equiv r)
equiv (Inv r) = equiv r
```

Figure 22: Equivalence relation induced by ordering relation

```
simplify :: Order t → Order t
simplify r@(NatO _) = r
simplify TrivO = TrivO
simplify (SumL r1 r2) = SumL (simplify r1) (simplify r2)
simplify (ProdL r1 r2) = ProdL (simplify r1) (simplify r2)
simplify (MapO f (MapO g r)) = simplify (MapO (g ∘ f) r)
simplify (MapO f r) = MapO f (simplify r)
simplify (ListL r) = ListL (simplify r)
simplify (BagO r) = BagO (simplify r)
simplify (SetO r) = SetO (simplify r)
simplify r@(Inv (NatO _)) = r
simplify (Inv TrivO) = TrivO
simplify (Inv (SumL r1 r2)) = sumR' (simplify (Inv r1)) (simplify (Inv r2))
simplify (Inv (ProdL r1 r2)) = ProdL (simplify (Inv r1)) (simplify (Inv r2))
simplify (Inv (MapO f (MapO g r))) = simplify (Inv (MapO (g ∘ f) r))
simplify (Inv (MapO f r)) = MapO f (simplify (Inv r))
simplify (Inv (ListL r)) = listR (simplify (Inv r))
simplify (Inv (BagO r)) = Inv (BagO (simplify r))
simplify (Inv (SetO r)) = Inv (SetO (simplify r))
simplify (Inv (Inv r)) = simplify r
```

Figure 23: Algebraic simplification of order representations

availability of high-bandwidth memory transfers and high-performance data-parallel primitives, and it makes sense to analyze complexity as a function of both input size N and word size w.[18]

In our setting the only meaningful measure of the input is its *size*: 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, where input size and number of input elements may be completely unrelated.

An apparently not widely known fact about comparison-based sorting algorithms—I have not seen it stated explicitly before— is that the complexity bounds in terms of N (size of input) and for n (number of input elements) are often the same, but *need* not be so: it depends on the complexity of the comparison function. (Recall that we are considering the case of sorting variable-sized elements.) In particular, an algorithm may not *necessarily* run in time $O(N \log N)$, even if it only executes $O(n \log n)$ comparison operations.

Theorem 14.1 *Let $(A, \leq)$ be a total preorder and assume that testing whether $v \leq w$ for elements v, w of size n, m, respectively, has time complexity $\Theta(\min\{n, m\})$ or $\Theta(n+m)$. Then comparison-based sorting algorithms have the worst-case time complexities given in Table 1 on a fixed-width RAM.*

PROOF (Proof sketch) For comparison functions executing in time $\Theta(n + m)$, that is in linear time and inspecting each bit in the two elements, 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.

For comparison functions executing in time $\Theta(\min\{n, m\})$, that is in linear time but only inspecting all the bits of the smaller of the two elements, it is easy to see that a worst-case input of size N consists of elements of same size k. In this case we have $N = n \cdot k$. Let $f(n)$ be the number of comparisons and constant time steps executed by a comparison-based sorting algorithm.

[18] In some analyses an "optimal" value for w as a function of N is chosen so as to provide a best possible worst-case complexity bound for an algorithm in terms of N alone. This identifies when word width and input size are perfectly matched. It is difficult to make practical sense of such an input dependent choice of word width, however, as it is impractical to choose the word width of the computer to run a program on only after the size of its input is available.

Table 1: Comparison-based sorting algorithms for complex data. Asymptotic worst-case running times in the size of the input, where comparison function is linear in the smaller, respectively larger of its two inputs

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

Note that $f(n) = \Omega(n \log n)$. The complexity of the algorithm in terms of N is $\Theta(f(\frac{N}{k}) \cdot k + f(\frac{N}{k}))$. The first summand counts the number of comparisons—note that each requires $\Theta(k)$ time— and the second summand counts the number of other steps. Thus we have $\Theta(f(\frac{N}{k}) \cdot (k + 1))$. Since f grows faster than $g(k) = k + 1$ we obtain the worst case for $k = 1$. In other words, constant-sized elements provide the worst-case scenario. . The complexity of a comparison-based sorting algorithm in terms of the size of the input is consequently $\Theta(f(N))$, which coincides with its complexity in terms of the number of comparison tests and other steps, assuming the latter each take constant time. □

Note that Mergesort and Heapsort require *quadratic time* for a comparison function that inspects all bits in its two inputs 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. If comparisons are on constant size data or are lexicographic string or list comparisons, both Mergesort and Heapsort run in time $\Theta(N \log N)$. This means that comparison-based sorting algorithms need to have their keys preprocessed by mapping them to constant-sized elements (e.g. `Int`) or to a list type under lexicographic ordering (e.g. `String`) to guarantee a $\Theta(N \log N)$ upper bound on the worst-case run time, which, luckily, is often possible.

14.3 Associative reshuffling

The code for both order and equivalence discrimination of products contains 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. Consider `sdisc`:

```
sdisc (ProdL r1 r2) xs   =
  [ vs | ys ← sdisc r1 [ (k1, (k2, v)) | ((k1, k2), v) ← xs ],
         vs ← sdisc 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, `ProdL (ProdL (ProdL r1 r2) r3) r4)`, with `r1, r2, r3, r4` being primitive order representations of the form `NatO n`. 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 `sdiscL` of the discriminator:

```
sdisc (ProdL r1 r2) xs   =
  [ vs | ys ← sdiscL r1 xs,
         vs ← sdisc r2 ys ]
```

But how to define `sdiscL` then? In particular, what to do when *its* argument in turn is a product representation? Introduce `sdiscLL`? Alternatively, 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 `sdiscA`, with the following clause for product orders:

```
sdiscA (ProdL r1 r2) f xs    =
  [ vs | ys ← sdiscA r1 (fst ∘ f) xs,
         vs ← sdiscA r2 (snd ∘ f) ys ]
```

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

In the end this can be extended to a generic definition of `sdiscA`, which actually sorts its input. It has one critical disadvantage, however: It has potentially 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 $(((...((k_1, k_2), k_3), ...), k_n), v)$, with $k_1, \ldots, k_n$ 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!

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

15 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 can be used both to partition data and also to sort them in linear time.

An important aspect of generic discriminators `sdisc`, `edisc` and, partially, `disc` is that they preserve abstraction: They provide observation of the order, respectively 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. It is of particular importance for heap-allocated garbage-collectable references. They can be represented as raw machine addresses or memory offsets and discriminated efficiently without breaking abstraction. No computation can observe anything about the particular machine address a reference has at any time. A discriminator can partition n constant-size elements in time $O(n)$. Using a binary equality test as the only operation to access the equivalence, this requires $\Omega(n^2)$ time. Fully abstract discriminators are principally superior for partitioning-like problems to both comparison functions and equality tests: they preserve abstraction, but provide asymptotically improved performance; and to hash functions: they match their algorithmic performance without compromising data abstraction.

15.1 Future work

It is quite easy to see how the definition of `sdisc` can be changed 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 representation 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 abstraction. 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 framework presented here can be developed into a framework that has a good chance of competing with even highly space-efficient in-place comparison-based sorting algorithms in most use scenarios of in-memory sorting. The naturally data-parallel comparison-free nature of discrimination may lend itself well to parallel computing architectures such as execution on GPGPUs, multicore architectures, and MapReduce-like cluster architectures (Dean and Ghemawat 2004).

Hinze[19] has observed that the generic order 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 equivalence discrimination can be extended to acyclic shared data structures. Using entirely different algorithmic techniques, equivalence discrimination can be extended to cyclic data 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, hash-free binary decision diagrams, reducing state graphs in model checkers, and the like.

The present functional specification of discrimination has been formulated in Haskell for clarity, not for performance beyond enabling some basic asymptotic reasoning and validating its principal viability. It performs

[19]Personal communication at IFIP TC2.8 Working Group meeting, Park City, Utah, June 15-22, 2008.

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 for truly high performance. 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.

To offer discrimination as a day-to-day programming tool, expressive and well-tuned libraries should be developed and evaluated empirically for usability and performance. Both functional languages such as Haskell, Standard ML, OCaml, Scheme, Erlang, and Clojure as well as popular languages such as C++, C#, Java, Python, and Visual Basic should be considered.

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. His papers are a veritable (if not always easily accessible) treasure trove of insights at the intersection algorithms and programming languages, the ramifications of which for programming and software construction have not been exhaustively explored yet.

Ralf Hinze alerted me to the possibility of employing GADTs for order representations by producing an implementation of generic discrimination (for standard orders) in Haskell after having seen a single presentation of it at the WG2.8 meeting in 2007.

Phil Wadler has provided detailed comments and criticisms. The addition of the column for $O(\min\{m, n\})$-time comparisons in Table 1 is a direct outgrowth of discussions with Phil. Phil, Torsten Grust and Janis Voigtländer provided valuable feedback specifically on the application of discrimination to join algorithms.

It has taken me several years and many iterations to generalize—and simultaneously distill—top-down 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 particularly thank Thomas Ambus, Martin Elsman, Hans Leiß, Ken Friis Larsen, Henning Niss, and Kristian Støvring.

The anonymous referees have greatly contributed to improving the original submission by finding and correcting infelicities and by suggesting improvements in technical content and presentation.

References

S. Abramsky and A. Jung. Domain theory. *Handbook of Logic in Computer Science, Semantic Structures*, 3:1–168, 1992.

A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

Amer Al-Badarneh and Fouad El-Aker. Efficient adaptive in-place radix sorting. *Informatica*, 15(3):295–302, 2004.

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 Anniual 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.

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.

Jon Bentley. Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721, 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/6424.315691.

J. Cai and R. Paige. Look ma, no hashing, and no arrays neither. In January, 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):189–228, July 1995.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, 2d edition edition, 2001. ISBN 0-262-03293-7 (MIT Press) and ISBN 0-07-013151-1 (McGraw-Hill).

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, San Francisco, California, December 2004.

Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359138.359142.

Gianni Franceschini, S. Muthukrishnan, and Mihai Pătraşcu. Radix sorting with no extra space. In *Proc. 15th European Symposium on Algorithms (ESA), Eilat, Israel*, volume 4698 of *Lecture Notes in Computer Science (LNCS)*, pages 194–205. Springer, October 2007.

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.

Yossi Gil and Yoav Zibin. Efficient algorithms for isomorphisms of simple types. *Mathematical Structures in Computer Science (MSCS)*, 15(05): 917–957, October 2005. doi: 10.1017/S0960129505004913.

Glasgow Haskell. *The Glasgow Haskell Compiler*, 2005. URL `http://www.haskell.org/ghc/`.

T. Grust, S. Sakr, and J. Teubner. XQuery on SQL hosts. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, page 263. VLDB Endowment, 2004.

Yijie Han and Mikkel Thorup. Integer sorting in $o(n\sqrt{\log\log n}$ expected time and linear space. In *Proceedings of the 43d Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–144. IEEE Computer Society, 2002.

Fritz Henglein. Multiset discrimination. Manuscript (incomplete), Department of Computer Science, University of Copenhagen (DIKU), September 2003. URL `file://localhost/Users/henglein/Documents/Research/papers/henglein2003.pdf`.

Fritz Henglein. Generic discrimination: Sorting and partitioning unshared data in linear time. In James Hook and Peter Thiemann, editors, *ICFP '08: Proceeding of the 13th ACM SIGPLAN*

international conference on Functional programming, pages 91–102, New York, NY, USA, September 2008. ACM. ISBN 978-1-59593-919-7. doi: http://doi.acm.org/10.1145/1411204.1411220. Nominated by ACM SIGPLAN for CACM Research Highlights (see http://sigplan.org/CACMPapers.htm).

Fritz Henglein. What is a sorting function? *J. Logic and Algebraic Programming (JLAP)*, 78(5):381–401, May-June 2009. doi: http://dx.doi.org/10.1016/j.jlap.2008.12.003. Invited submisstion to special issue on 19th Nordic Workshop on Programming Theory (NWPT).

Fritz Henglein. Optimizing relational algebra operations using discrimination-based joins and lazy products. In *Proc. ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation*, pages 73–82, New York, NY, USA, January 18-19 2010. ACM. ISBN 978-1-60558-727-1. doi: http://doi.acm.org/10.1145/1706356.1706372. Also DIKU TOPPS D-report no. 611.

Fritz Henglein and Ken Friis Larsen. Generic multiset programming for language-integrated querying. In *Proc. 6th Workshop on Generic Programming (WGP)*, September 2010.

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.

Paul Hudak, John Peterson, and Joseph H. Fasel. A gentle introduction to Haskell Version 98. Online tutorial, May 1999. URL `http://www.haskell.org/tutorial`.

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 Netherlands, 2008. doi: http://dx.doi.org/10.1007/978-1-4020-6585-9. URL `http://www.springer.com/computer/programming/book/978-1-4020-6584-2`. ISBN 978-1-4020-6584-2 (Print), 978-1-4020-6585-9 (Online).

J. P. Jouannaud and P. Lescanne. On multiset orderings. *Information Processing Letters*, 25(2):57–63, 1982.

Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

Arne Maus. ARL, a faster in-place, cache friendly sorting algorithm. In *Norwegian Informatics Conference (NIK), Kongsberg, Norway*. Tapir, 2002. ISBN 82-91116-45-8.

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 B. Pehrson and I. Simon, editors, *Proc. 13th World Computer Congress, Vol. 1*. Elsevier Science B.V. (North Holland), 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.

Simon Peyton Jones. The Haskell 98 language. *J. Functional Programming (JFP)*, 13(1):0–146, January 2003.

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 J. Oudshoorn, editor, *ACSC*, volume 16 of *CRPIT*, pages 11–18. Australian Computer Society, 2003. ISBN 0-909-92594-1. doi: http://crpit.com/confpapers/CRPITV16Sinha.pdf.

C. Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49, 2000. ISSN 1388-3690.

R. Tarjan. *Data Structures and Network Flow Algorithms*, volume CMBS 44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.

Phil Trinder and Philip Wadler. List comprehensions and the relational calculus. In *Proceedings of the 1988 Glasgow Workshop on Functional Programming, Rothesay, Scotland*, pages 115–123, August 1988.

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. 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 160–171. ACM, ACM Press, January 2003. SIGPLAN Notices, Vol. 38, No. 1.