Sorting and Searching by Distribution: From Generic Discrimination to Generic Tries

Fritz Henglein[1] and Ralf Hinze[2]

[1] Department of Computer Science, University of Copenhagen
henglein@diku.dk
[2] Department of Computer Science, University of Oxford
ralf.hinze@cs.ox.ac.uk

Abstract. A discriminator partitions values associated with keys into groups listed in ascending order. Discriminators can be defined generically by structural recursion on representations of ordering relations. Employing type-indexed families we demonstrate how tries with an optimal-time lookup function can be constructed generically in worst-case linear time. We provide generic implementations of comparison, sorting, discrimination and trie building functions and give equational proofs of correctness that highlight core relations between these algorithms.

1 Introduction

Sorting and searching are some of the most fundamental topics in computer science. In this paper we define *generic* functions for solving sorting and searching problems, based on *distributive*, that is "radix-sort-like", techniques rather than comparison-based techniques. The functions are indexed by representations of ordering relations on keys of type K. In each case the input is an association list of key-value pairs, and the values are treated as *satellite data*, that is, the functions are *parametric* in the value type V. Intuitively, this means values are pointers that are not dereferenced during execution of these functions [1]. We identify a hierarchy of operations:[1]

$$\begin{aligned} &\mathit{sort} &&:: \mathit{Order}\ k \to [k \times v] \to [v] \\ &\mathit{discr} &&:: \mathit{Order}\ k \to [k \times v] \to [[v]] \\ &\mathit{trie} &&:: \mathit{Order}\ k \to [k \times v] \to \mathsf{Trie}\ k\ [v] \end{aligned}$$

The *sorting function*, *sort*, outputs the value components according to the given order on K without, however, returning the key component. For example,

$$\begin{aligned} &\gg \mathit{sort}\ (\mathit{OList}\ \mathit{OChar})\ [(\texttt{"ab"}, 1), (\texttt{"ba"}, 2), (\texttt{"abc"}, 3), (\texttt{"ba"}, 4)] \\ &[1, 3, 2, 4]\ , \end{aligned}$$

[1] Executable code is rendered in Haskell, which requires lower-case identifiers for type variables. We use the corresponding upper-case identifiers in the running text and in program calculations.

C.-c. Shan (Ed.): APLAS 2013, LNCS 8301, pp. 315–332, 2013.

where *OList OChar* denotes the standard lexicographic order on strings. We require that *sort* be stable in the sense that the relative order of values with equivalent keys is preserved. Discarding the keys may seem surprising and restrictive at first. Nothing is lost, however, since parametricity allows us to arrange it so that the keys are also returned. We simply associate the keys with themselves.

$\gg$ *sort* (*OList OChar*) [("ab", "ab"), ("ba", "ba"), ("abc", "abc"), ("ba", "ba")]
["ab", "abc", "ba", "ba"]

The *discriminator*, *discr*, outputs the value components grouped into runs of values associated with equivalent keys. For example,

$\gg$ *discr* (*OList OChar*) [("ab", 1), ("ba", 2), ("abc", 3), ("ba", 4)]
[[1], [3], [2, 4]] .

The *trie constructor*, *trie*, outputs a trie that can subsequently be efficiently searched for values associated to a particular key. The type of trie constructed depends on the type of the keys. For example,

$\gg$ **let** t = *trie* (*OList OChar*) [("ab", 1), ("ba", 2), ("abc", 3), ("ba", 4)]
$\gg$ *lookup* t "ba"
Just [2, 4] .

The function *discr* was introduced by Henglein [2,3] (originally called *sdisc*). It provides a framework for bootstrapping any base sorting algorithm for a *finite* type, such as bucket sort, to a large class of user-definable orders on first-order and recursive types. To this end it employs a strategy corresponding to most-significant-digit (MSD) in radix sorting.

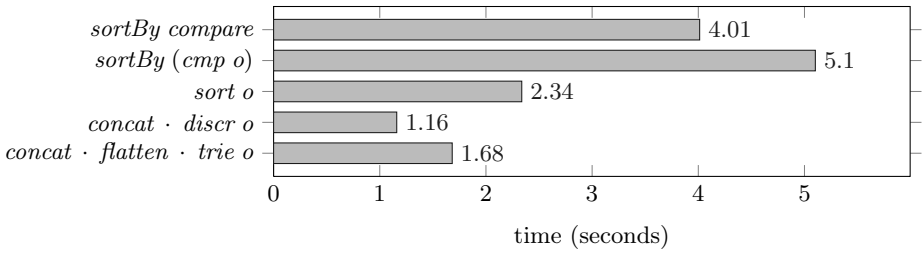
The functions *sort* and *trie* are novel. Algorithmically, *sort* does the same as *discr*, but employing a least-significant-digit (LSD) strategy. Drawing on the informal correspondence of MSD radix sort with tries [4, p. 3], *trie* generalizes *discr* and generates the generalized tries introduced by Hinze [5]. It subsumes *discr* (which in turn subsumes *sort*) in the sense that it executes in the same time (usually linear in the size of the input keys), but additionally facilitates efficient search for values associated with any key.

In this paper we make the following novel contributions:

- We show that a function of type $[K \times V] \to [V]$ is a *stable sorting function* if and only if it is *strongly natural* in V, preserves singleton lists, and sorts lists of length 2 correctly. A function is strongly natural if it commutes with filtering, that is, the removal of elements from a list.
- We give new *generic* definitions of: *sort*, which generalizes least-significant-digit (LSD) radix sort to arbitrary types and orders definable by an expressive language of *order representations*; and *trie*, which generalizes *discr* to construct efficiently key-searchable tries. Both run in worst-case linear time for a large class of orders.
- We provide equational proofs for *sort o* being a stable sorting function and show that *sort o* = *concat* $\cdot$ *discr o* and *discr o* = *flatten* $\cdot$ *trie o* for all

inductively defined order representations o, where *concat* is list concatenation and *flatten* lists the values stored in a trie in ascending key order. The first equality is nontrivial as *discr* and *sort* have different underlying algorithmic strategies for product types: MSD versus LSD. The proof highlights the strong naturality properties of *sort* and *discr*.
– We offer preliminary benchmark results of our generic distributive sorting functions, which are surprisingly promising when compared to Haskell's built-in comparison-based sorting function.

The paper focuses on and highlights the core relations between these algorithms, notably the role of *strong naturality*. Here we limit ourselves to a restricted class of orders and leave asymptotic analysis, performance engineering, and a proper empirical performance analysis for future work. But certainly some benchmarks are not amiss to whet the appetite. The task is to sort the words of Project Gutenberg's The Bible, King James Version (5218802 characters, 824337 words). We compare Haskell's built-in *sortBy* called with Haskell's own *compare* and our generically defined comparison function *cmp o*, to generic sorting and generic discrimination, and to sorting via generic tries.



We assume familiarity with the programming language GHC Haskell and basic notions of category theory. Unless noted otherwise, we work in **Set**, the category of sets and total functions.

2 Order Representations

Comparison-based sorting and searching methods are attractive because they easily generalize to arbitrary orders: simply parameterize the program code for, say, Quicksort [6] over its comparison function, and apply it to a user-defined ordering $leq :: T \to T \to \mathbb{B}$. An analogous approach works for searching on T using, say, red-black trees [7,8]. While maximally expressive, specifying orders via such "black-box" binary comparisons, has two disadvantages:

1. Deliberately or erroneously, *leq* may not implement a total preorder.
2. Both sorting and searching are subject to lower bounds on their performance: sorting requires $\Omega(n \log n)$ comparisons, and searching for a key requires $\Omega(\log n)$, where n is the number of keys in the input.

However, theoretically and practically faster *distributive* methods are known for *certain* orders, notably radix sorting and tries.

As shown by Henglein [3], many orders can be denoted by *order representations*, constructors for building new orders from old:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Order} :: * \to * \ \textbf{where}\\
&\quad \mathit{OUnit} :: \mathit{Order}\ ()\\
&\quad \mathit{OSum} :: \mathit{Order}\ k_1 \to \mathit{Order}\ k_2 \to \mathit{Order}\ (k_1 + k_2)\\
&\quad \mathit{OProd} :: \mathit{Order}\ k_1 \to \mathit{Order}\ k_2 \to \mathit{Order}\ (k_1 \times k_2)\\
&\quad \mathit{OMap} :: (k_1 \to k_2) \to (\mathit{Order}\ k_2 \to \mathit{Order}\ k_1)\\
&\quad \mathit{OChar} :: \mathit{Order}\ \mathit{Char} \quad \text{-- 7 bit ASCII .}
\end{aligned}
$$

Here *OUnit* denotes the trivial order on the unit type. $\mathit{OSum}\ o_1\ o_2$ represents the lexicographic order on tagged values such that *Inl*-tagged values are less than *Inr*-tagged values, and values with the same tag are ordered by o_1 or o_2, depending on the tag. $\mathit{OProd}\ o_1\ o_2$ denotes the lexicographic order on pairs, ordering pairs according to their first component, where pairs with equivalent first component are ordered according to their second component. $\mathit{OMap}\ f\ o$ orders the domain of f according to the order o on its codomain. Note that *OMap* is contravariant. Finally, *OChar* denotes the standard order on 7-bit ASCII characters.

The *OMap*-constructor adds considerable expressiveness. For example,

$$
\begin{aligned}
&\mathit{rprod} :: \mathit{Order}\ k_1 \to \mathit{Order}\ k_2 \to \mathit{Order}\ (k_1 \times k_2)\\
&\mathit{rprod}\ o_1\ o_2 = \mathit{OMap}\ (\lambda(a, b) \to (b, a))\ (\mathit{OProd}\ o_2\ o_1)
\end{aligned}
$$

specifies the lexicographic order on pairs based on the *second* component as the dominant one. Similarly, *rsum* can be specified, which orders *Inr*-tagged values as less than *Inl*-tagged ones.

Order representations are terms that can be treated *inductively* as finite trees or *coinductively* as potentially infinite trees.

The coinductive approach permits definition of orders for recursive data types by guarded recursion. For example,

$$
\begin{aligned}
&\mathit{olist} :: \mathit{Order}\ k \to \mathit{Order}\ [k]\\
&\mathit{olist}\ o = os\ \textbf{where}\ os = \mathit{OMap}\ \mathit{out}\ (\mathit{OSum}\ \mathit{OUnit}\ (\mathit{OProd}\ o\ os))\\
&\mathit{out} :: [a] \to () + a \times [a]\\
&\mathit{out}\ [\,] \quad\ = \mathit{Inl}\ ()\\
&\mathit{out}\ (a : as) = \mathit{Inr}\ (a, as)
\end{aligned}
$$

defines the standard lexicographic order on lists, based on the element order o. Henglein [3] takes the coinductive approach with additional order constructors for inverse, multiset and set orders.

In the inductive approach we can add new constructors explicitly:

$$\mathit{OList} :: \mathit{Order}\ k \to \mathit{Order}\ [k]$$

or, more generally, employ an explicit fixed point operator [2].

The expressiveness of order representations is orthogonal to the aims of this paper. For simplicity we assume the inductive approach and concentrate on sums and products.

3 Generic Comparison

We require order representations to denote *total preorders*.

$$
\begin{aligned}
&\mathit{leq} :: \mathit{Order}\ k \to (k \to k \to \mathbb{B}) \\
&\mathit{leq}\ \mathit{OUnit}\ a\ b &&= \mathit{True} \\
&\mathit{leq}\ (\mathit{OSum}\ o_1\ o_2)\ a\ b &&= \mathbf{case}\ (a, b)\ \mathbf{of} \\
&&&\quad (\mathit{Inl}\ a_1, \mathit{Inl}\ a_2) \to \mathit{leq}\ o_1\ a_1\ a_2 \\
&&&\quad (\mathit{Inl}\ \_, \ \mathit{Inr}\ \_) \to \mathit{True} \\
&&&\quad (\mathit{Inr}\ \_, \ \mathit{Inl}\ \_) \to \mathit{False} \\
&&&\quad (\mathit{Inr}\ b_1, \mathit{Inr}\ b_2) \to \mathit{leq}\ o_2\ b_1\ b_2 \\
&\mathit{leq}\ (\mathit{OProd}\ o_1\ o_2)\ a\ b &&= \mathit{leq}\ o_1\ (\mathit{fst}\ a)\ (\mathit{fst}\ b) \wedge \\
&&&\quad (\mathit{leq}\ o_1\ (\mathit{fst}\ b)\ (\mathit{fst}\ a) \Longrightarrow \mathit{leq}\ o_2\ (\mathit{snd}\ a)\ (\mathit{snd}\ b)) \\
&\mathit{leq}\ (\mathit{OMap}\ g\ o)\ a\ b &&= \mathit{leq}\ o\ (g\ a)\ (g\ b) \\
&\mathit{leq}\ (\mathit{OChar})\ a\ b &&= a \leqslant b
\end{aligned}
$$

leq o is indeed a total preorder; it is transitive and total, $\mathit{leq}\ o\ x\ y \vee \mathit{leq}\ o\ y\ x$. Because of totality the case for *OProd* can also be written as

$$
\begin{aligned}
\mathit{leq}\ (\mathit{OProd}\ o_1\ o_2)\ a\ b = \mathbf{if}\ &\mathit{leq}\ o_2\ (\mathit{snd}\ a)\ (\mathit{snd}\ b)\ \mathbf{then}\ \mathit{leq}\ o_1\ (\mathit{fst}\ a)\ (\mathit{fst}\ b) \\
&\mathbf{else}\ \ \neg\ (\mathit{leq}\ o_1\ (\mathit{fst}\ b)\ (\mathit{fst}\ a))
\end{aligned}
$$

This variant is strict (*leq* o_1 and *leq* o_2 are called), but it calls *leq* o_1 only once. The first variant is lazy (*leq* o_2 is not necessarily called), but possibly calls *leq* o_1 twice.

The function *leq* implements a two-way comparison; a more useful function is $\mathit{cmp} :: \mathit{Order}\ k \to (k \to k \to \mathit{Ordering})$, which implements a three-way comparison and avoids the double traversal in the product case.

4 Generic Distributive Sorting

Generic sorting takes a list of key-value pairs and returns the values in non-decreasing order of their associated keys. The keys are discarded in the course of this process. The idea is that, barring *OMap* in order representations, each component of each key is touched *exactly* once. Consequently, the running time of *sort* is proportional to the total size of the keys (again, ignoring *OMap*).

$$
\begin{aligned}
&\mathit{sort} :: \mathit{Order}\ k \to [k \times v] \to [v] \\
&\mathit{sort}\ o &&[\,] &&= [\,] \\
&\mathit{sort}\ (\mathit{OUnit}) &&\mathit{rel} &&= \mathit{map}\ \mathit{val}\ \mathit{rel} \\
&\mathit{sort}\ (\mathit{OSum}\ o_1\ o_2) &&\mathit{rel} &&= \mathit{sort}\ o_1\ (\mathit{filter}\ \mathit{froml}\ \mathit{rel}) +\!\!+ \mathit{sort}\ o_2\ (\mathit{filter}\ \mathit{fromr}\ \mathit{rel}) \\
&\mathit{sort}\ (\mathit{OProd}\ o_1\ o_2) &&\mathit{rel} &&= \mathit{sort}\ o_1\ (\mathit{sort}\ o_2\ (\mathit{map}\ \mathit{curryr}\ \mathit{rel})) \\
&\mathit{sort}\ (\mathit{OMap}\ g\ o) &&\mathit{rel} &&= \mathit{sort}\ o\ (\mathit{map}\ (g \times \mathit{id})\ \mathit{rel}) \\
&\mathit{sort}\ (\mathit{OChar}) &&\mathit{rel} &&= \mathit{bucketSort}\ (\texttt{'\textbackslash NUL'}, \texttt{'\textbackslash DEL'})\ \mathit{rel}
\end{aligned}
$$

Like generic comparison, *sort* is indexed by order representations. It is furthermore parametric in the type of values. Let us discuss each case in turn.

The first equation is vital for the coinductive approach to recursive types. It is necessary to ensure that for each recursive invocation of *sort* the total size of the keys is strictly decreasing.

For the unit type there is little to do: we simply discard the keys using *val* defined $val\,(k, v) = v$.

The case for sums takes an approach à la Quicksort: the input list is partitioned into a list whose keys are of the form $Inl\,k_1$ and a second list whose keys are of the form $Inr\,k_2$. The constructors are discarded, the sub-lists are sorted recursively using the appropriate orders, and the final results are concatenated. (As an aside, in the partitioning phase we touch the keys actually twice, but this is easily avoided by combining the two sweeps into a single one.) The function $filter :: (a \to \mathsf{Maybe}\ b) \to ([a] \to [b])$, called *mapMaybe* elsewhere, combines mapping and filtering: if the argument function returns a value of the form $Just\ b$, then b is included in the output list. If the result is *Nothing*, well, nothing is added. (Don't confuse our *filter* with Haskell's $filter :: (a \to \mathbb{B}) \to ([a] \to [a])$).

$$\begin{aligned} &filter :: (a \to \mathsf{Maybe}\ b) \to ([a] \to [b]) \\ &filter\ p\ xs = [y \mid x \leftarrow xs, Just\ y \leftarrow [p\ x]] \end{aligned}$$

The function $froml :: (k_1 + k_2 \times v) \to \mathsf{Maybe}\ (k_1 \times v)$ maps $(Inl\ k_1, v)$ to $Just\ (k_1, v)$, and $(Inr\ k_2, v)$ to *Nothing*. The function *fromr* is defined analogously.

The most interesting case is the one for products. The natural isomorphism $curryr : (K_1 \times K_2) \times V \cong K_2 \times (K_1 \times V)$ shifts the *more significant* part of the key into the value component. Then *sort* is called twice: the first invocation sorts according to o_2 discarding the K_2 part, the second sorts according to o_1 discarding the K_1 component. For this to be correct, $sort\ o_1$ had better be stable; we shall return to this point below. Furthermore, *sort* relies on *polymorphic recursion*: the first call to *sort* instantiates V to $K_1 \times V$.

For *OMap* we simply apply the key transformation using $map\ (g \times id)$ and then sort the transformed keys-value pairs.

Characters are sorted using bucket sort, which can be seen as a specialization of *sort* (actually of *discr* introduced in Section 5) for enumeration types.

$$\begin{aligned} &bucketSort :: (Bounded\ i, Ix\ i) \Rightarrow (i, i) \to [(i, v)] \to [v] \\ &bucketSort\ bs\ rel = concat\ (elems\ (accumArray\ (\lambda ws\ w \to ws +\!\!+ [w])\ [\,]\ bs\ rel)) \end{aligned}$$

(Here $+\!\!+$ is used for clarity; in our implementation it is replaced by a constant-time operation.) Any other algorithm for sorting characters or, for that matter, other primitive types could be used. The particular algorithm invoked can even be made *data dependent*; for small *rel* we might choose insertion sort instead of bucket sort to avoid sparse bucket table traversals. The key point is that *sort* reduces a sorting problem to basic sorting on finite domains. Conversely, it extends distributive sorting from their restricted domains such as small integers and character strings to arbitrary orders definable by order representations.

Let us now turn to the correctness of *sort*. Its implementation builds upon standard components, except perhaps the case for sums, which relies on the function *filter*. Note that *filter* takes a partial function as an argument, represented

by a total function of type $A \to \mathsf{Maybe}\ B$, an arrow in the Kleisli category induced by the monad Maybe. For the proofs it will be convenient to actually work in this Kleisli category (but only for the arguments of *filter*). In the calculations, we signal these steps by the hint "Kleisli:". A few remarks are in order.

Working in the Kleisli category has the advantage that the notation is fairly light-weight: we write *id* rather than η, and we write $p \cdot q$ for the Kleisli composition of p and q rather than $\mu \cdot \mathsf{Maybe}\ p \cdot q$. We also silently embed total functions into the Kleisli category: *filter f* really means *filter* $(\eta \cdot f)$. In any case, there is little room for confusion since we let $f, g \ldots$ range over total functions and p, q over partial functions. The product $\times$ can be lifted to a binary functor $\otimes$ over the Kleisli category, which is, however, not a categorical product. Rather, $\otimes$ is a so-called tensor product, a binary functor which is coherently associative and commutative. We overload $\times$ to denote both the product in the underlying category and the tensor product.

A partial function that we will use time and again is inl°, the left-inverse of *inl*. The partial function *froml* can be neatly expressed in terms of inl°: we have $froml = inl^\circ \otimes id$, or just $inl^\circ \times id$. Likewise, $fromr = inr^\circ \otimes id$.

The function *filter* satisfies a variety of properties. First and foremost, it is a monoid homomorphism:

$$filter\ p\ [\,] = [\,]\ , \tag{4.1a}$$

$$filter\ p\ (xs +\!\!+ ys) = filter\ p\ xs +\!\!+ filter\ p\ ys\ . \tag{4.1b}$$

Furthermore, *filter* is functorial, taking Kleisli arrows to arrows in the underlying category. Formally, *filter* is the arrow part of the functor $\mathsf{Filter} : \mathbf{Kleisli} \to \mathbf{Set}$, whose object part is defined $\mathsf{Filter}\ A = \mathsf{List}\ A$. In other words, *filter* preserves identity and composition.

$$filter\ id = id \tag{4.2a}$$

$$filter\ (p \cdot q) = filter\ p \cdot filter\ q \tag{4.2b}$$

Moreover, if its first argument is a total function, then *filter f* is just $\mathsf{List}\ f$.

Most of the following proofs will be conducted in a point-free style. For reference, here is a suitably reworked version of *sort*.

$$\begin{aligned} sort\ (OUnit) &= \mathsf{List}\ val \\ sort\ (OSum\ o_1\ o_2) &= sort\ o_1 \cdot filter\ (inl^\circ \times id) +\!\!+ sort\ o_2 \cdot filter\ (inr^\circ \times id) \\ sort\ (OProd\ o_1\ o_2) &= sort\ o_1 \cdot sort\ o_2 \cdot \mathsf{List}\ curryr \end{aligned}$$

The sum case uses a lifted variant of append, $(f +\!\!+ g)\ x = f\ x +\!\!+ g\ x$, overloading the operator $+\!\!+$ to denote both the lifted and the unlifted version.

4.1 Naturality

A vital property of *sort o* is that it is natural in the type of values, $sort\ o : \mathsf{List} \circ (K \times) \dot{\to} \mathsf{List}$, that is

$$\mathsf{List}\ f \cdot sort\ o = sort\ o \cdot \mathsf{List}\ (id \times f)\ , \tag{4.3}$$

for all $f : A \to B$.

Because of naturality it is sufficient to show that the instance $sort\ o{:}[K{\times}\mathbb{N}] \to [\mathbb{N}]$ works correctly. (Recall that values can be seen as pointers; natural numbers are like unique pointers.) Formally, *sort o* is fully determined by this instance:

$$\begin{array}{ll} & sort\ o\ [(k_1, v_1), \ldots, (k_n, v_n)] \\ = & \quad \{\ \text{let } ix : \mathbb{N} \to V \text{ be an indexing function so that } ix\ i = k_i\ \} \\ & sort\ o\ (\mathsf{List}\ (id \times ix)\ [(k_1, 1), \ldots, (k_n, n)]) \\ = & \quad \{\ sort \text{ is natural (4.3)}\ \} \\ & \mathsf{List}\ ix\ (sort\ o\ [(k_1, 1), \ldots, (k_n, n)])\ . \end{array}$$

In the last equation *sort o* is used at instance $\mathbb{N}$.

Of course, the statement that *sort o* is natural requires proof. Actually, *sort* satisfies a much stronger property, which we discuss next.

4.2 Strong Naturality

Property (4.3) remains valid if we replace List by *filter*:

$$filter\ p \cdot sort\ o = sort\ o \cdot filter\ (id \times p)\ , \tag{4.4}$$

for all $p : A \to \mathsf{Maybe}\ B$. It does not matter whether we filter before or after an invocation of *sort o*, as long as *filter* only refers to the values, and not the keys.

Now, since *filter* is the arrow part of a functor between the Kleisli category of Maybe and the underlying category, (4.4) also amounts to a naturality property, $sort\ \ o : \mathsf{Filter} \circ (K \otimes\ ) \dot{\to} \mathsf{Filter}$. A simple consequence of what we call strong naturality (4.4) is that *sort* preserves the empty list.

Turning to the proof of (4.4), we proceed by induction over the structure of order representations.

Case $o = OUnit$:

$$\begin{array}{ll} & filter\ p \cdot sort\ OUnit \\ = & \quad \{\ \text{definition of } sort\ \} \\ & filter\ p \cdot \mathsf{List}\ val \\ = & \quad \{\ \text{Kleisli: } p \cdot val = val \cdot (id \times p)\ \} \\ & \mathsf{List}\ val \cdot filter\ (id \times p) \\ = & \quad \{\ \text{definition of } sort\ \} \\ & sort\ OUnit \cdot filter\ (id \times p) \end{array}$$

Recall that $\times$ aka $\otimes$ is *not* a categorical product in the Kleisli category, we have $val \cdot (id \times p) = p \cdot val$, but *not* $key \cdot (id \times p) = id \cdot key$ where $key\ (k, v) = k$.

Case $o = OSum\ o_1\ o_2$: the central step is the fourth one, where we swap two filters, one that acts on the keys and a second that acts on the values.

$$\begin{array}{ll} & filter\ p \cdot sort\ (OSum\ o_1\ o_2) \\ = & \quad \{\ \text{definition of } sort\ \} \\ & filter\ p \cdot (sort\ o_1 \cdot filter\ (inl^\circ \times id) +\!\!+ \ldots) \\ = & \quad \{\ filter \text{ is a monoid homomorphism (4.1b)}\ \} \\ & filter\ p \cdot sort\ o_1 \cdot filter\ (inl^\circ \times id) +\!\!+ \ldots \\ = & \quad \{\ \text{ex hypothesi}\ \} \end{array}$$

$$
\begin{array}{ll}
 & \mathit{sort}\ o_1 \cdot \mathit{filter}\ ((id + id) \times p) \cdot \mathit{filter}\ (\mathit{inl}^\circ \times id) \mathbin{+\!\!+} \ldots \\
= & \quad \{\ \text{Kleisli: } \times \text{ is a binary functor}\ \} \\
 & \mathit{sort}\ o_1 \cdot \mathit{filter}\ (\mathit{inl}^\circ \times id) \cdot \mathit{filter}\ ((id + id) \times p) \mathbin{+\!\!+} \ldots \\
= & \quad \{\ \text{fusion: } f \cdot h \mathbin{+\!\!+} g \cdot h = (f \mathbin{+\!\!+} g) \cdot h\ \} \\
 & (\mathit{sort}\ o_1 \cdot \mathit{filter}\ (\mathit{inl}^\circ \times id) \mathbin{+\!\!+} \ldots) \cdot \mathit{filter}\ ((id + id) \times p) \\
= & \quad \{\ \text{definition of } \mathit{sort}\ \} \\
 & \mathit{sort}\ (\mathit{OSum}\ o_1\ o_2) \cdot \mathit{filter}\ ((id + id) \times p)
\end{array}
$$

Note that the lifting of $+$ is a categorical coproduct in the Kleisli category.
Case $o = \mathit{OProd}\ o_1\ o_2$: note that Property (4.4) is universally quantified over all p. This is essential as *sort* relies on polymorphic recursion: the second use of the induction hypothesis $(*)$ instantiates p to $id \times p$.

$$
\begin{array}{ll}
 & \mathit{filter}\ p \cdot \mathit{sort}\ (\mathit{OProd}\ o_1\ o_2) \\
= & \quad \{\ \text{definition of } \mathit{sort}\ \} \\
 & \mathit{filter}\ p \cdot \mathit{sort}\ o_1 \cdot \mathit{sort}\ o_2 \cdot \mathsf{List}\ \mathit{curryr} \\
= & \quad \{\ \text{ex hypothesi}\ \} \\
 & \mathit{sort}\ o_1 \cdot \mathit{filter}\ (id \times p) \cdot \mathit{sort}\ o_2 \cdot \mathsf{List}\ \mathit{curryr} \\
= & \quad \{\ \text{ex hypothesi } (*)\ \} \\
 & \mathit{sort}\ o_1 \cdot \mathit{sort}\ o_2 \cdot \mathit{filter}\ (id \times (id \times p)) \cdot \mathsf{List}\ \mathit{curryr} \\
= & \quad \{\ \text{Kleisli: } (q \times (p \times r)) \cdot \mathit{curryr} = \mathit{curryr} \cdot ((p \times q) \times r)\ \} \\
 & \mathit{sort}\ o_1 \cdot \mathit{sort}\ o_2 \cdot \mathsf{List}\ \mathit{curryr} \cdot \mathit{filter}\ ((id \times id) \times p) \\
= & \quad \{\ \text{definition of } \mathit{sort}\ \} \\
 & \mathit{sort}\ (\mathit{OProd}\ o_1\ o_2) \cdot \mathit{filter}\ ((id \times id) \times p)
\end{array}
$$

The natural isomorphisms $\mathit{val}{:}1 \times K \cong K$ and $\mathit{curryr}{:}(K_1 \times K_2) \times V \cong K_2 \times (K_1 \times V)$ are also natural isomorphisms in the Kleisli category.

4.3 Correctness: Permutation

Our first goal is to show that *sort o* produces a permutation of the input values. Perhaps surprisingly, it suffices to show that *sort o* permutes one-element lists! We already know that it is sufficient to show the correctness of a particular instance: $\mathit{sort}\ o\ [(k_1, 1), \ldots, (k_n, n)]$. Now, let $\wr i \int$ be the partial function that maps i to i and is undefined otherwise. Let $1 \leqslant i \leqslant n$, then

$$
\begin{array}{ll}
 & \mathit{filter}\ \wr i \int\ (\mathit{sort}\ o\ [(k_1, 1), \ldots, (k_n, n)]) \\
= & \quad \{\ \mathit{sort} \text{ is strongly natural (4.4)}\ \} \\
 & \mathit{sort}\ o\ (\mathit{filter}\ (id \times \wr i \int)\ [(k_1, 1), \ldots, (k_n, n)]) \\
= & \quad \{\ \mathit{filter}\ (id \times \wr i \int)\ [(k_1, 1), \ldots, (k_n, n)] = [(k_i, i)]\ \} \\
 & \mathit{sort}\ o\ [(k_i, i)] \\
= & \quad \{\ \textit{proof obligation: } \mathit{sort} \text{ permutes 1-element lists (4.5)}\ \} \\
 & [i]\ .
\end{array}
$$

Thus, *sort o* outputs each index exactly once; in other words, it permutes the input list. The proof obligation (recall that $\mathit{return}\ a = [a]$)

$$\mathit{sort}\ o \cdot \mathit{return} = \mathit{return} \cdot \mathit{val} \tag{4.5}$$

is easy to discharge and left as an instructive exercise to the reader.

4.4 Correctness: Ordered

Our second task is to show that the values are output in non-decreasing order of their associated keys. We aim to show

$$sort\ o\ [\ldots,(k_i,i),\ldots,(k_j,j),\ldots] = [\ldots,i,\ldots,j,\ldots] \iff leq\ o\ k_i\ k_j\ .$$

Due to strong naturality, it suffices to show that *sort o* works correctly on two-element lists! Let $\wr i,j\int$ be the partial function that maps i to i and j to j and is undefined otherwise. Let $1 \leqslant i,j \leqslant n$, then

$$\begin{array}{ll} & filter\ \wr i,j\int\ (sort\ o\ [(k_1,1),\ldots,(k_n,n)]) = [i,j] \\ \iff & \{\ sort \text{ is strongly natural (4.4)}\ \} \\ & sort\ o\ (filter\ (id \times \wr i,j\int)\ [(k_1,1),\ldots,(k_n,n)]) = [i,j] \\ \iff & \{\ filter\ (id \times \wr i,j\int)\ [(k_1,1),\ldots,(k_n,n)] = [(k_i,i),(k_j,j)]\ \} \\ & sort\ o\ [(k_i,i),(k_j,j)] = [i,j] \\ \iff & \{\ \textit{proof obligation: } sort \text{ sorts 2-element lists (4.6)}\ \} \\ & leq\ o\ k_i\ k_j\ . \end{array}$$

Thus, *sort o* outputs i before j if and only if *leq o* k_i k_j. Since we already know that *sort o* permutes its input, this implies the correctness of *sort o*.

It remains to show that *sort* treats 2-element lists correctly: let $i \neq j$, then

$$sort\ o\ [(a,i),(b,j)] = [i,j] \iff leq\ o\ a\ b\ . \tag{4.6}$$

Since only a small finite number of cases have to be considered, this is a simple exercise, which we relegate to Appendix A.

5 Generic Discrimination

A discriminator returns a list of non-empty lists of values, where the inner lists group values whose keys are equivalent. Again, the keys are discarded in the process, but, barring *OMap* in order representations, this time each component of each key is touched *at most* once.

$$\begin{array}{lll} discr :: Order\ k \to [k \times v] \to [[v]] & & \\ discr\ o & [\,] & = [\,] \\ discr\ o & [(k,v)] & = [[v]] \\ discr\ OUnit & rel & = [map\ val\ rel] \\ discr\ (OSum\ o_1\ o_2) & rel & = discr\ o_1\ (filter\ froml\ rel) +\!\!+ discr\ o_2\ (filter\ fromr\ rel) \\ discr\ (OProd\ o_1\ o_2) & rel & = concat\ (map\ (discr\ o_2)\ (discr\ o_1\ (map\ curryl\ rel))) \\ discr\ (OMap\ g\ o) & rel & = discr\ o\ (map\ (g \times id)\ rel) \\ discr\ (OChar) & rel & = bucketDiscr\ (\texttt{'\textbackslash NUL'}, \texttt{'\textbackslash DEL'})\ rel \end{array}$$

In the unit case we have identified a group of key-value pairs whose keys are equivalent, even identical. This group is returned as a singleton list. The sum case is the same as for *sort*. The most interesting case is again the one for products.

The natural isomorphism $curryl : (K_1 \times K_2) \times V \cong K_1 \times (K_2 \times V)$ also known as *assoc* shifts the *least significant* part of the key into the value component. The resulting list is discriminated according to o_1, each of the resulting groups is discriminated according to o_2, and finally the nested groups are flattened. The definition of *discr* has an additional base case for the singleton list. This may improve the performance dramatically since the key component in the argument need not be traversed. For lexicographic string sorting this specializes to the property of MSD radix sort, which only traverses the *minimum distinguishing prefixes* of the strings, which may be substantially fewer characters than their total number. Finally, characters are sorted using bucket sort—this time we simply return the list of *non-empty* buckets.

$$
\begin{aligned}
&\mathit{bucketDiscr} :: (\mathit{Bounded}\ i, \mathit{Ix}\ i) \Rightarrow (i, i) \to [i \times v] \to [[v]] \\
&\mathit{bucketDiscr}\ \mathit{bs}\ \mathit{rel} \\
&\quad = [\mathit{xs} \mid \mathit{xs} \leftarrow \mathit{elems}\ (\mathit{accumArray}\ (\lambda \mathit{ws}\ w \to \mathit{ws} +\!\!+ [w])\ [\,]\ \mathit{bs}\ \mathit{rel}), \neg\ (\mathit{null}\ \mathit{xs})]
\end{aligned}
$$

As for *sort*, any other base type discriminator could be plugged in.

5.1 Correctness

If we concatenate the groups returned by a generic discriminator, we obtain generic sorting.

$$\mathit{concat} \cdot \mathit{discr}\ o = \mathit{sort}\ o \tag{5.1}$$

The proof is straightforward for units and sums, the interesting case is again the one for products. For products discrimination works from left to right, whereas sorting proceeds right to left. This means we have to be able to swap operations:

$$
\begin{aligned}
& \mathit{concat} \cdot \mathit{discr}\ (\mathit{OProd}\ o_1\ o_2) \\
= \quad & \{\ \text{definition of } \mathit{discr}\ \} \\
& \mathit{concat} \cdot \mathit{concat} \cdot \mathsf{List}\ (\mathit{discr}\ o_2) \cdot \mathit{discr}\ o_1 \cdot \mathsf{List}\ \mathit{curryl} \\
= \quad & \{\ \text{monad law}\ \} \\
& \mathit{concat} \cdot \mathsf{List}\ \mathit{concat} \cdot \mathsf{List}\ (\mathit{discr}\ o_2) \cdot \mathit{discr}\ o_1 \cdot \mathsf{List}\ \mathit{curryl} \\
= \quad & \{\ \text{ex hypothesi}\ \} \\
& \mathit{concat} \cdot \mathsf{List}\ (\mathit{sort}\ o_2) \cdot \mathit{discr}\ o_1 \cdot \mathsf{List}\ \mathit{curryl} \\
= \quad & \{\ \textit{proof obligation:}\ \text{see below}\ \} \\
& \mathit{concat} \cdot \mathit{discr}\ o_1 \cdot \mathit{sort}\ o_2 \cdot \mathsf{List}\ \mathit{curryr} \\
= \quad & \{\ \text{ex hypothesi}\ \} \\
& \mathit{sort}\ o_1 \cdot \mathit{sort}\ o_2 \cdot \mathsf{List}\ \mathit{curryr} \\
= \quad & \{\ \text{definition of } \mathit{sort}\ \} \\
& \mathit{sort}\ (\mathit{OProd}\ o_1\ o_2)\ .
\end{aligned}
$$

The property used in the central step, being able to swap *sort* o_2 and *discr* o_1, is actually not specific to *sort*. Our generic discriminators commute with *every strong* natural transformation. Let $\mathit{discr}\ o : [K \times V] \to [[V]]$ and $\pi : [A \times V] \to [V]$. If π is strongly natural, $\mathit{filter}\ p \cdot \pi = \pi \cdot \mathit{filter}\ (\mathit{id} \times p)$, then

$$\mathsf{List}\ \pi \cdot \mathit{discr}\ o = \mathit{discr}\ o \cdot \pi \cdot \mathsf{List}\ \mathit{swap}\ , \tag{5.2}$$

where $swap : K_1 \times (K_2 \times V) \cong K_2 \times (K_1 \times V)$. Note that $swap \cdot curryl = curryr$.
Case $o = OUnit$:

$$
\begin{array}{ll}
 & \mathsf{List}\ \pi \cdot discr\ OUnit\ :\ [1 \times (A \times V)] \to [[\,V]] \\
= & \quad \{\ \text{definition of } discr\ \} \\
 & \mathsf{List}\ \pi \cdot return \cdot \mathsf{List}\ val \\
= & \quad \{\ return \text{ is natural: } \mathsf{List}\ f \cdot return = return \cdot f\ \} \\
 & return \cdot \pi \cdot \mathsf{List}\ val \\
= & \quad \{\ \text{Kleisli: } val = (id \times val) \cdot swap\ \} \\
 & return \cdot \pi \cdot \mathsf{List}\ (id \times val) \cdot \mathsf{List}\ swap \\
= & \quad \{\ \textit{assumption:}\ \pi \text{ is (strongly) natural}\ \} \\
 & return \cdot \mathsf{List}\ val \cdot \pi \cdot \mathsf{List}\ swap \\
= & \quad \{\ \text{definition of } discr\ \} \\
 & discr\ OUnit \cdot \pi \cdot \mathsf{List}\ swap
\end{array}
$$

Case $o = OSum\ o_1\ o_2$:

$$
\begin{array}{ll}
 & \mathsf{List}\ \pi \cdot discr\ (OSum\ o_1\ o_2)\ :\ [(K_1 + K_2) \times (A \times V)] \to [[\,V]] \\
= & \quad \{\ \text{definition of } discr\ \} \\
 & \mathsf{List}\ \pi \cdot (discr\ o_1 \cdot filter\ (inl^\circ \times id) +\!\!+ \ldots) \\
= & \quad \{\ \mathsf{List}\ \pi \text{ is a monoid homomorphism}\ \} \\
 & \mathsf{List}\ \pi \cdot discr\ o_1 \cdot filter\ (inl^\circ \times id) +\!\!+ \ldots \\
= & \quad \{\ \text{ex hypothesi}\ \} \\
 & discr\ o_1 \cdot \pi \cdot \mathsf{List}\ swap \cdot filter\ (inl^\circ \times id) +\!\!+ \ldots \\
= & \quad \{\ \text{Kleisli: } swap \cdot (p \times (q \times r)) = (q \times (p \times r)) \cdot swap\ \} \\
 & discr\ o_1 \cdot \pi \cdot filter\ (id \times (inl^\circ \times id)) \cdot \mathsf{List}\ swap +\!\!+ \ldots \\
= & \quad \{\ \textit{assumption:}\ \pi \text{ is strongly natural}\ \} \\
 & discr\ o_1 \cdot filter\ (inl^\circ \times id) \cdot \pi \cdot \mathsf{List}\ swap +\!\!+ \ldots \\
= & \quad \{\ \text{fusion: } f \cdot h +\!\!+ g \cdot h = (f +\!\!+ g) \cdot h\ \} \\
 & (discr\ o_1 \cdot filter\ (inl^\circ \times id) +\!\!+ \ldots) \cdot \pi \cdot \mathsf{List}\ swap \\
= & \quad \{\ \text{definition of } discr\ \} \\
 & discr\ (OSum\ o_1\ o_2) \cdot \pi \cdot \mathsf{List}\ swap
\end{array}
$$

Case $o = OProd\ o_1\ o_2$:

$$
\begin{array}{ll}
 & \mathsf{List}\ \pi \cdot discr\ (OProd\ o_1\ o_2)\ :\ [(K_1 \times K_2) \times (A \times V)] \to [[\,V]] \\
= & \quad \{\ \text{definition of } discr\ \} \\
 & \mathsf{List}\ \pi \cdot concat \cdot \mathsf{List}\ (discr\ o_2) \cdot discr\ o_1 \cdot \mathsf{List}\ curryl \\
= & \quad \{\ concat \text{ is natural: } \mathsf{List}\ f \cdot concat = concat \cdot \mathsf{List}\ (\mathsf{List}\ f)\ \} \\
 & concat \cdot \mathsf{List}\ (\mathsf{List}\ \pi \cdot discr\ o_2) \cdot discr\ o_1 \cdot \mathsf{List}\ curryl \\
= & \quad \{\ \text{ex hypothesi}\ \} \\
 & concat \cdot \mathsf{List}\ (discr\ o_2 \cdot \pi \cdot \mathsf{List}\ swap) \cdot discr\ o_1 \cdot \mathsf{List}\ curryl \\
= & \quad \{\ discr\ o \text{ is natural: } \mathsf{List}\ (\mathsf{List}\ f) \cdot discr\ o = discr\ o_1 \cdot \mathsf{List}\ (id \times f)\ \} \\
 & concat \cdot \mathsf{List}\ (discr\ o_2 \cdot \pi) \cdot discr\ o_1 \cdot \mathsf{List}\ ((id \times swap) \cdot curryl) \\
= & \quad \{\ \text{ex hypothesi}\ \} \\
 & concat \cdot \mathsf{List}\ (discr\ o_2) \cdot discr\ o_1 \cdot \pi \cdot \mathsf{List}\ (swap \cdot (id \times swap) \cdot curryl) \\
= & \quad \{\ \text{Kleisli: } swap \cdot (id \times swap) \cdot curryl = (id \times curryl) \cdot swap\ \}
\end{array}
$$

$$
\begin{aligned}
& \mathit{concat} \cdot \mathsf{List}\ (\mathit{discr}\ o_2) \cdot \mathit{discr}\ o_1 \cdot \pi \cdot \mathsf{List}\ ((\mathit{id} \times \mathit{curryl}) \cdot \mathit{swap}) \\
= \quad & \{\ \textit{assumption: } \pi \text{ is (strongly) natural}\ \} \\
& \mathit{concat} \cdot \mathsf{List}\ (\mathit{discr}\ o_2) \cdot \mathit{discr}\ o_1 \cdot \mathsf{List}\ \mathit{curryl} \cdot \pi \cdot \mathsf{List}\ \mathit{swap} \\
= \quad & \{\ \text{definition of } \mathit{discr}\ \} \\
& \mathit{discr}\ (\mathit{OProd}\ o_1\ o_2) \cdot \pi \cdot \mathsf{List}\ \mathit{swap}
\end{aligned}
$$

Of course, *discr o* itself is also strongly natural:

$$\mathsf{List}\ (\mathit{filter}\ p) \cdot \mathit{discr}\ o = \mathit{discr}\ o \cdot \mathit{filter}\ (\mathit{id} \times p)\ , \tag{5.3}$$

for all $p : A \to \mathsf{Maybe}\ B$. The proof is similar to the one for *sort*.

6 Generic Distributive Searching

Let us now turn to distributive searching using tries. In this paper we concentrate on bulk operations such as *trie* and *flatten*. One-at-a-time operations such as *lookup* and *insert* have been described elsewhere [5]. It turns out that *trie* is very similar to *discr*—we essentially replace $+\!\!+$ and *concat* by trie constructors. This move retains more of the original information, which is vital for supporting subsequent *efficient random* access to the values associated with a key. By storing the keys together with the values in the trie, we can even recreate all of the original keys. (If *OMap*'s key transformations are injective this can even be done without explicitly storing the keys.)

For every order representation there is a corresponding trie constructor. Additionally, we have an empty trie, which is important for efficiency reasons [5].

```
data Trie :: * → * → * where
  TEmpty :: Trie k v
  TUnit  :: v → Trie () v
  TSum   :: Trie k1 v → Trie k2 v → Trie (k1 + k2) v
  TProd  :: Trie k1 (Trie k2 v) → Trie (k1, k2) v
  TMap   :: (k1 → k2) → (Trie k2 v → Trie k1 v)
  TChar  :: Char.Trie v → Trie Char v
```

A trie of type $\mathsf{Trie}\ K\ V$ represents a finite mapping from K to V, sometimes written V^K. The cases for unit, sums, and products are based on the law of exponentials: $V^1 \cong V$, $V^{K_1+K_2} \cong V^{K_1} \times V^{K_2}$, and $V^{K_1 \times K_2} \cong (V^{K_2})^{K_1}$. The second but last case is interesting: the counterpart of *OMap* is *TMap*, which retains the key transformation. This is necessary when searching for a key that is subject to an *OMap*-order. Finally, we assume the existence of a suitable library, *Char*, implementing finite maps with character keys; for instance, character-indexed arrays, simple lists, binary trees (the basis of ternary tries). Indeed, depending on the actual data encountered, multiple data structures may even be mixed.

A trie for a given key type is a functor.

```
instance Functor (Trie k) where
  fmap f (TEmpty)   = TEmpty
  fmap f (TUnit v)  = TUnit (f v)
```

$$
\begin{aligned}
&\mathit{fmap}\ f\ (\mathit{TSum}\ t_1\ t_2) &&= \mathit{TSum}\ (\mathit{fmap}\ f\ t_1)\ (\mathit{fmap}\ f\ t_2)\\
&\mathit{fmap}\ f\ (\mathit{TProd}\ t) &&= \mathit{TProd}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ t)\\
&\mathit{fmap}\ f\ (\mathit{TMap}\ g\ t) &&= \mathit{TMap}\ g\ (\mathit{fmap}\ f\ t)\\
&\mathit{fmap}\ f\ (\mathit{TChar}\ t) &&= \mathit{TChar}\ (\mathit{fmap}\ f\ t)
\end{aligned}
$$

The operation *flatten* lists the values stored in a trie.

$$
\begin{aligned}
&\mathit{flatten} :: \mathsf{Trie}\ k\ v \to [v]\\
&\mathit{flatten}\ (\mathit{TEmpty}) &&= [\,]\\
&\mathit{flatten}\ (\mathit{TUnit}\ v) &&= [v]\\
&\mathit{flatten}\ (\mathit{TSum}\ t_1\ t_2) &&= \mathit{flatten}\ t_1 +\!\!+ \mathit{flatten}\ t_2\\
&\mathit{flatten}\ (\mathit{TProd}\ t) &&= \mathit{concatMap}\ \mathit{flatten}\ (\mathit{flatten}\ t)\\
&\mathit{flatten}\ (\mathit{TMap}\ g\ t) &&= \mathit{flatten}\ t\\
&\mathit{flatten}\ (\mathit{TChar}\ t) &&= \mathit{Char.flatten}\ t
\end{aligned}
$$

It is natural in V, that is, $\mathit{flatten} : \mathsf{Trie}\ K \dot{\to} \mathsf{List}$.

The operation *trie* turns a finite relation, represented by an association list of type $[K \times V]$, into a finite list-valued map, represented by a trie of type $\mathsf{Trie}\ K\ [V]$.

$$
\begin{aligned}
&\mathit{trie} :: \mathit{Order}\ k \to [k \times v] \to \mathsf{Trie}\ k\ [v]\\
&\mathit{trie}\ o &&[\,] &&= \mathit{TEmpty}\\
&\mathit{trie}\ \mathit{OUnit} &&\mathit{rel} &&= \mathit{TUnit}\ (\mathit{map}\ \mathit{val}\ \mathit{rel})\\
&\mathit{trie}\ (\mathit{OSum}\ o_1\ o_2) &&\mathit{rel} &&= \mathit{TSum}\ (\mathit{trie}\ o_1\ (\mathit{filter}\ \mathit{froml}\ \mathit{rel}))\ (\mathit{trie}\ o_2\ (\mathit{filter}\ \mathit{fromr}\ \mathit{rel}))\\
&\mathit{trie}\ (\mathit{OProd}\ o_1\ o_2) &&\mathit{rel} &&= \mathit{TProd}\ (\mathit{fmap}\ (\mathit{trie}\ o_2)\ (\mathit{trie}\ o_1\ (\mathit{map}\ \mathit{curryl}\ \mathit{rel})))\\
&\mathit{trie}\ (\mathit{OMap}\ g\ o) &&\mathit{rel} &&= \mathit{TMap}\ g\ (\mathit{trie}\ o\ (\mathit{map}\ (g \times \mathit{id})\ \mathit{rel}))\\
&\mathit{trie}\ (\mathit{OChar}) &&\mathit{rel} &&= \mathit{TChar}\ (\mathit{Char.trie}\ \mathit{rel})
\end{aligned}
$$

As we have noted before, *trie* arises out of *discr* by replacing $+\!\!+$, *concat* etc by the appropriate trie constructors.

Indeed, if we 'undo' the transformation using *flatten*, we obtain the generic discriminator.

$$\mathit{discr}\ o = \mathit{flatten} \cdot \mathit{trie}\ o \tag{6.1}$$

The straightforward proof can be found in Appendix B.

7 Related Work

Drawing on the algorithmic techniques termed *multiset discrimination* developed by Paige and others [9], Henglein [2,3] has shown how to make MSD distributive sorting generic by introducing generic discriminators, which have linear-time performance over a rich class of orders.

Building on the work of Connelly and Morris [10], Hinze [5] pioneered the type-indexed tries we produce. Here they are extended to support orders defined with *OMap*. The generic LSD distributive sorting and trie building functions developed here are also new. In particular, *trie* constructs a trie in bulk without incurring the substantial update costs of one-by-one insertion.

Wadler [11] derives as a "free theorem" that any function parametric in binary comparison $\leqslant$ commutes with *map* f if the function f is an order embedding, $f\,x \leqslant' f\,y \iff x \leqslant y$; this includes all comparison-based sorting algorithms. As shown by Day et al. [12], Knuth's *0-1 principle* for sorting networks [13] can also be seen as a free theorem for sorting networks formulated as comparator-parametrized functions. These properties correspond to naturality, respectively parametricity properties on the keys; they are *different* from our naturality property 4.3 since the latter applies to the value components and leaves the key components invariant.

Our strong naturality property 4.4, coupled with preserving singletons and correct sorting of two-element lists, corresponds to Henglein's *consistent permutativity*, which characterizes stable sorting functions [14]. It is, however, a more general and a more elegant formulation supporting equational reasoning. In particular, it highlights the semantic benefits of adopting a formulation for sorting based on key-value pairs rather than keys alone.

Gibbons [15] shows how an LSD radix sort for lists can be derived from a stable MSD radix sort that first builds an explicit trie and then flattens it into the result list. Since MSD radix sort, even with explicit tries, is sometimes preferable to LSD radix sort (to avoid sparse bucket table traversal [16] and for large data sets [17]), the derivation makes sense in both directions. Our development can be seen as a generalization of Gibbons' work: it works for arbitrary denotable orders over any type; we decompose MSD sorting into discrimination followed by concatenation, without the need for a trie (though it can be achieved by way of *trie*); and our commutativity property 5.2 holds for *any* strong natural transformation, not just for sorting functions.

8 Conclusion

Comparison-based sorting algorithms and search trees are easily made generic, that is, applicable to user-defined orders, by abstraction over the comparison function. This has arguably contributed to their popularity even though distributive (radix/trie) and hashing techniques often have superior performance for special types, such as machine integers and character strings.

We have shown how to construct generic comparison, sorting, discrimination and trie building operations by induction over a class of orders including standard orders on primitive types; lexicographic orders on sums, products and lists; and orders defined as the inverse image of a given order under an arbitrary function.

We have identified strong naturality—commutativity with filtering—as a powerful property of stable sorting functions, and shown discrimination to commute with any strongly natural transformation, including, but not limited to, stable sorting functions.

The trie building operation yields a data structure that is not only asymptotically as efficient as discrimination but also supports efficient key-based random access, without incurring a one-at-a-time insertion overhead during construction.

Future work consists of extending equational reasoning and calculational correctness proofs to coinductive order representations; investigating *data-dependent* variations of our generic functions and staged execution for our *data-independent* generic functions by compile-time specialization (partial evaluation) and exploiting parallelism at word, multicore, and manycore/GPU levels; and eventually providing architecture-independent frameworks encapsulating distributive sorting and searching methods as semantically (obeying representation independence) and computationally (exhibiting superior performance) well-behaved alternatives to comparison-based methods.

Acknowledgements. We would like to thank the anonymous referees of APLAS 2013 and Nicolas Wu for suggesting various presentational improvements. We owe a particular debt of gratitude to Richard Bird for carefully reading a draft version of this paper, pointing out typographical errors, glitches of language, and for suggesting pronounceable identifiers. This work has been funded by EPSRC grant number EP/J010995/1 and Danish Research Council grant 10-092299 for HIPERFIT.

References

1. Strachey, C.: Fundamental concepts in programming languages. Higher-order and Symbolic Computation 13(1), 11–49 (2000)
2. Henglein, F.: Generic discrimination: Sorting and partitioning unshared data in linear time. In: Hook, J., Thiemann, P. (eds.) Proc. 13th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP), pp. 91–102. ACM (September 2008)
3. Henglein, F.: Generic top-down discrimination for sorting and partitioning in linear time. Journal of Functional Programming (JFP) 22(3), 300–374 (2012)
4. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: SODA 1997: Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 360–369. Society for Industrial and Applied Mathematics, Philadelphia (1997)
5. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming 10(4), 327–351 (2000)
6. Hoare, C.A.: Quicksort. The Computer Journal 5(1), 10–16 (1962)
7. Bayer, R.: Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Informatica 1(4), 290–306 (1972)
8. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: Proc. 19th Annual Symposium on Foundations of Computer Science (FOCS), pp. 8–21. IEEE (1978)
9. Cai, J., Paige, R.: Using multiset discrimination to solve language processing problems without hashing. Theoretical Computer Science (TCS) 145(1-2), 189–228 (1995)
10. Connelly, R.H., Morris, F.L.: A generalization of the trie data structure. Mathematical Structures in Computer Science 5(3), 381–418 (1995)
11. Wadler, P.: Theorems for free! In: Proc. Functional Programming Languages and Computer Architecture (FPCA), pp. 347–359. ACM Press, London (1989)

12. Day, N.A., Launchbury, J., Lewis, J.: Logical abstractions in Haskell. In: Proc. Haskell Workshop. Number UU-CS-1999-28 in Technical Report, Utrecht, The Netherlands, Utrecht University (1999)
13. Knuth, D.: The Art of Computer Programming: Sorting and Searching, 2nd edn., vol. 3. Addison Wesley (1998)
14. Henglein, F.: What is a sorting function? J. Logic and Algebraic Programming (JLAP) 78(5), 381–401 (2009)
15. Gibbons, J.: A pointless derivation of radix sort. Journal of Functional Programming 9(3), 339–346 (1999)
16. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal of Computing 16(6), 973–989 (1987)
17. Sinha, R., Zobel, J.: Efficient trie-based sorting of large sets of strings. In: Proc. 26th Australasian Computer Science Conference (ACSC), pp. 11–18 (2003)

A Proof of Property (4.6)

The cases for unit and sums are straightforward. We only consider the product case, which is actually instructive.
Case $o = OProd\ o_1\ o_2$: again, we make use of naturality to be able to apply the induction assumption.

$$
\begin{array}{ll}
 & sort\ (OProd\ o_1\ o_2)\ [((a_1,a_2),i),((b_1,b_2),j)] = [i,j] \\
\iff & \{\ \text{definition of } sort\ \} \\
 & sort\ o_1\ (sort\ o_2\ (\mathsf{List}\ curryr\ [((a_1,a_2),i),((b_1,b_2),j)])) = [i,j] \\
\iff & \{\ \text{definition of } curryr\ \} \\
 & sort\ o_1\ (sort\ o_2\ [(a_2,(a_1,i)),(b_2,(b_1,j))]) = [i,j] \\
\iff & \{\ \text{let } re\ i = (a_1,i) \text{ and } re\ j = (b_1,j)\ \} \\
 & sort\ o_1\ (sort\ o_2\ (\mathsf{List}\ (id \times re)\ [(a_2,i),(b_2,j)]) = [i,j] \\
\iff & \{\ sort \text{ is natural (4.3)}\ \} \\
 & sort\ o_1\ (\mathsf{List}\ re\ (sort\ o_2\ [(a_2,i),(b_2,j)]) = [i,j]
\end{array}
$$

The strict version of *leq* suggests to conduct a case analysis on $leq\ o_2\ a_2\ b_2$.

Case $\boldsymbol{leq\ o_2\ a_2\ b_2}$**:**

$$
\begin{array}{ll}
\iff & \{\ \text{ex hypothesi}\ \} \\
 & sort\ o_1\ (\mathsf{List}\ re\ [i,j]) = [i,j] \\
\iff & \{\ \text{definition of } re\ \} \\
 & sort\ o_1\ [(a_1,i),(b_1,j)] = [i,j] \\
\iff & \{\ \text{ex hypothesi}\ \} \\
 & leq\ o_1\ a_1\ b_1 \\
\iff & \{\ \text{definition of } leq\ \} \\
 & leq\ (OProd\ o_1\ o_2)\ (a_1,a_2)\ (b_1,b_2)
\end{array}
$$

Case $\neg\ (\boldsymbol{leq\ o_2\ a_2\ b_2})$**:**

$$
\begin{array}{ll}
\iff & \{\ \text{ex hypothesi}\ \} \\
 & sort\ o_1\ (\mathsf{List}\ re\ [j,i]) = [i,j] \\
\iff & \{\ \text{definition of } re\ \} \\
 & sort\ o_1\ [(b_1,j),(a_1,i)] = [i,j] \\
\iff & \{\ \text{ex hypothesi}\ \} \\
 & \neg\ (leq\ o_1\ b_1\ a_1) \\
\iff & \{\ \text{definition of } leq\ \} \\
 & leq\ (OProd\ o_1\ o_2)\ (a_1,a_2)\ (b_1,b_2)
\end{array}
$$

B Proof of $\mathit{discr}\ o = \mathit{flatten} \cdot \mathit{trie}\ o$ (6.1)

Case $o = \mathit{OUnit}$:

$$
\begin{aligned}
& \mathit{flatten} \cdot \mathit{trie}\ \mathit{OUnit} \\
= \quad & \{ \text{ definition of } \mathit{trie} \ \} \\
& \mathit{flatten} \cdot \mathit{TUnit} \cdot \mathit{map}\ \mathit{val} \\
= \quad & \{ \text{ definition of } \mathit{flatten} \ \} \\
& \mathit{return} \cdot \mathit{map}\ \mathit{val} \\
= \quad & \{ \text{ definition of } \mathit{discr} \ \} \\
& \mathit{discr}\ \mathit{OUnit}
\end{aligned}
$$

Case $o = \mathit{OSum}\ o_1\ o_2$:

$$
\begin{aligned}
& \mathit{flatten} \cdot \mathit{trie}\ (\mathit{OSum}\ o_1\ o_2) \\
= \quad & \{ \text{ definition of } \mathit{trie} \ \} \\
& \mathit{flatten} \cdot \mathit{TSum} \cdot (\mathit{trie}\ o_1 \cdot \mathit{filter}\ (\mathit{inl}^\circ \times \mathit{id}) \vartriangle \ldots) \\
= \quad & \{ \text{ definition of } \mathit{flatten} \ \} \\
& (\mathit{flatten} \cdot \mathit{outl} +\!\!+ \ldots) \cdot (\mathit{trie}\ o_1 \cdot \mathit{filter}\ (\mathit{inl}^\circ \times \mathit{id}) \vartriangle \ldots) \\
= \quad & \{ \text{ fusion and computation } \} \\
& \mathit{flatten} \cdot \mathit{trie}\ o_1 \cdot \mathit{filter}\ (\mathit{inl}^\circ \times \mathit{id}) +\!\!+ \ldots \\
= \quad & \{ \text{ ex hypothesi } \} \\
& \mathit{discr}\ o_1 \cdot \mathit{filter}\ (\mathit{inl}^\circ \times \mathit{id}) +\!\!+ \ldots \\
= \quad & \{ \text{ definition of } \mathit{discr} \ \} \\
& \mathit{discr}\ (\mathit{OSum}\ o_1\ o_2)
\end{aligned}
$$

Case $o = \mathit{OProd}\ o_1\ o_2$: here we make essential use of the fact that $\mathsf{Trie}\ K$ is a functor and that *flatten* is natural in V.

$$
\begin{aligned}
& \mathit{flatten} \cdot \mathit{trie}\ (\mathit{OProd}\ o_1\ o_2) \\
= \quad & \{ \text{ definition of } \mathit{trie} \ \} \\
& \mathit{flatten} \cdot \mathit{TProd} \cdot \mathsf{Trie}\ K_1\ (\mathit{trie}\ o_2) \cdot \mathit{trie}\ o_1 \cdot \mathsf{List}\ \mathit{curryl} \\
= \quad & \{ \text{ definition of } \mathit{flatten} \ \} \\
& \mathit{concat} \cdot \mathsf{List}\ \mathit{flatten} \cdot \mathit{flatten} \cdot \mathsf{Trie}\ K_1\ (\mathit{trie}\ o_2) \cdot \mathit{trie}\ o_1 \cdot \mathsf{List}\ \mathit{curryl} \\
= \quad & \{ \ \mathit{flatten} \text{ is natural: } \mathsf{List}\ f \cdot \mathit{flatten} = \mathit{flatten} \cdot \mathsf{Trie}\ K\ f \ \} \\
& \mathit{concat} \cdot \mathsf{List}\ \mathit{flatten} \cdot \mathsf{List}\ (\mathit{trie}\ o_2) \cdot \mathit{flatten} \cdot \mathit{trie}\ o_1 \cdot \mathsf{List}\ \mathit{curryl} \\
= \quad & \{ \text{ ex hypothesi, twice } \} \\
& \mathit{concat} \cdot \mathsf{List}\ (\mathit{discr}\ o_2) \cdot \mathit{discr}\ o_1 \cdot \mathsf{List}\ \mathit{curryl} \\
= \quad & \{ \text{ definition of } \mathit{discr} \ \} \\
& \mathit{discr}\ (\mathit{OProd}\ o_1\ o_2) \ .
\end{aligned}
$$