Distributive Sorting and Searching

From Generic Discrimination to Generic Tries

Fritz Henglein; Ralf Hinze

DIKU, University of Copenhagen; DCS, University of Oxford

2013-12-16

SAPLING 2013
Macquarie University, Australia

Sorting and searching

- Two principal approaches:
  - Comparison-based methods (e.g. Quicksort; red-black trees)
  - Distributive methods (e.g. radix sort; tries, hashing)
- *Generic* sorting and searching?
  - Parameter: User-defined sort order

Comparison-based sorting: Quicksort

```
qsort :: List String -> List String
qsort []         =  []
qsort (x : xs)   =  qsort littles
                         ++ [x]
                         ++ qsort bigs
    where  littles    =  [ a | a <- xs, a <=  x ]
           bigs       =  [ a | a <- xs, a >   x ]
```

Generic comparison-based sorting: HOF abstraction

```
qsortBy :: (k -> k -> Bool) -> List k -> List k
qsortBy (<=)  []          =  []
qsortBy (<=)  (x : xs)    =  qsortBy (<=) littles
                             ++ [x]
                             ++ qsortBy (<=) bigs
   where  littles     =  [ a | a <- xs, a <=  x ]
          bigs        =  [ a | a <- xs, not (a <= x) ]
```

Generic comparison-based sorting: Discussion

- Methods are easily made generic: turn the comparison function into a parameter ("black-box" approach).
- *But*:
- User-specified function may or may not be a *comparison* function.
- Both sorting and searching are subject to lower bounds:
  - sorting requires $\Omega(n \log n)$ comparisons, and
  - searching for a key requires $\Omega(\log n)$ comparisons,

  where n is the number of keys in the input.
- Often, comparison is *not* a constant-time operation

Idea: DSL for orders

Order Representations: Definition

- An element of `Order K` *represents* an order over the type `K`:

```
data Order :: * -> where
  OUnit ::Order()
  OSum  :: Order k1 -> Order k2 -> Order (k1 + k2)
  OProd :: Order k1 -> Order k2 -> Order (k1, k2)
  OMap  :: (k1 -> k2) -> (Order k2 -> Order k1)
  OChar :: Order Char -- 7 bit ASCIIdata
```

Order Representations: Examples

- Reverse lexicographic order:

```
rprod  :: Order k1 -> Order k2 -> Order (k1, k2)
rprod o1 o2 = OMap (fn (a, b) -> (b, a)) (OProd o2 o1)
```

- Ordering recursive types, eg strings:

```
data String = [] | (Char : String)

ostring :: Order String
ostring = OMap out (OSum OUnit (OProd OChar ostring))

out :: String -> () + (Char, String)
out []         =  Inl ()
out (a : as)   =  Inr (a, as)
```

Generic comparison function

- Interprets an order representation as comparison function:

```
lte ::  Order k -> (k -> k -> Bool)
lte OUnit a b          =  True
lte (OSum o1 o2) a b =
    case (a, b) of
         (Inl a1,  Inl a2  ) -> lte o1 a1 a2
         (Inl _,   Inr _   ) -> True
         (Inr _,   Inl _   ) -> False
         (Inr b1,  Inr b2  ) -> lte o2 b1 b2
lte (OProd o1 o2) a b  =
    lte o1 (fst a) (fst b) &&
    (lte o1 (fst b) (fst a) ==> lte o2 (snd a) (snd b))
lte (OMap g o) a b     =  lte o (g a) (g b)
lte (OChar) a b        =  a <= b
```

- Sorting lists of words:

```
qsort (lte ostring)
```

Distributive sorting & searching: Idea

- Employ the structure of order representations *directly*.
- A hierarchy of operations:

```
sort  :: Order k -> List (k,v) -> List v
discr :: Order k -> List (k,v) -> List (List v)
trie  :: Order k -> List (k,v) -> Trie k (List v)
```

- We separate keys from *satellite data*, i.e. associated values.

Distributive Sorting & Searching: Examples

- The keys are discarded:

  `sort ostring [("ab",1), ("ba",2), ("abc",3), ("ba",4)]`

  $\Longrightarrow$ `[1,3,2,4]`

  Note: `sort` is stable.
- Returning the keys (sorting as permutation):

  `sort ostring (map (fn a -> (a, a)) ["ab","ba","abc","ba"])`

  $\Longrightarrow$ `["ab","abc","ba","ba"]`
- Grouping values with equivalent keys:

  `discr ostring [("ab",1), ("ba",2), ("abc",3), ("ba",4)]`

  $\Longrightarrow$ `[[1],[3],[2,4]]`
- Distributive searching:

```
let dict =
    trie ostring [("ab",1), ("ba",2), ("abc",3), ("ba",4)]
in lookup dict "ba"
```

  $\Longrightarrow$ `Just [2,4]`

Generic distributive sorting

`sort o` takes list of key-value pairs, returns values in non-decreasing order of their associated keys.

```
sort :: Order k -> List (k,v) -> List v
sort o              []  =  []
sort OUnit          rel =  map (fn (k,v) -> v) rel
sort (OSum o1 o2)   rel =
  sort o1 (filter froml rel) ++ sort o2 (filter fromr rel)
sort (OProd o1 o2) rel =
  sort o1 (sort o2 (map curryr rel))
sort (OMap g o)     rel =
  sort o (map (f * id) rel)
sort (OChar)        rel =  bucketsort rel
```

Let us look at some clauses.

Distributive sorting: Discussion

- Each component of each key is touched *exactly* once.
  - Ignoring `OMap`.
- The running time is *linear* in the *total size* of the keys.
- `sort` generalizes *least-significant-digit (LSD) radix sort* to user-definable orders on arbitrary data types.
- `sort` uses o as a control structure to reduce a sorting problem to basic sorting on finite domains (bootstrapping).
  - Practical performance determined by sorting small integers.

Distributive sorting: Properties

- *Naturality*, `sort o` commutes with `map`:

  `map f . sort o = sort o . map (id * f)`

- *Strong naturality*, `sort o` commutes with filtering:

  `filter p . sort o = sort o . filter (id * p)`

- Sorting singletons

  `sort o [(k, v)] = [v]`

- Sorting pairs:
  `sort o [(a,v), (b,w)] = [v,w]` $\Longleftrightarrow$ `lte o a b = True`

Theorem: Strong naturality + sorting singletons + sorting pairs $\Longrightarrow$ stable sort.

Generic Tries: Definition

- An element of `Trie K V` represents a *finite map* from `K` to `V`. Introduce *map constructors*:

```
data Trie k v where
 TEmpty :: Trie k v            -- empty map
 TUnit  :: v -> Trie () v      -- singleton map
 TSum   :: Trie k1 -> 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.Map v -> Trie Char v
```

- The first type argument is an *index*, the second a *parameter*.

Building tries in bulk

```
build :: Order k -> List (k, v) -> Trie k (List v)
build o              []  = TEmpty
build OUnit          rel = TUnit (map val rel)
build (OSum o1 o2)   rel = TSum (build o1 (filter froml rel))
                                (build o2 (filter fromr rel))
build (OProd o1 o2) rel =
      TProd (fmap (build o2) (build o1 (map curryl rel)))
build (OMap g o)     rel = TMap g (build o (map (g * id) rel))
build (OChar)        rel = TChar (Char.build rel)
```

where

```
curryl ((k1, k2), v) = (k1, (k2, v))
```

and

```
fmap :: (v -> w) -> Trie k v -> Trie k w
```

is morphism mapping component of functor `Trie k`

Building tries in bulk: Complexity

`trie` and `lookup` are *asymptotically optimal*:

- `trie` builds a trie in time *linear* in the total size of the keys in the input.
- `lookup :: Trie k v -> k -> Maybe v` returns its result in time *linear* in the size of the key input (independent of the trie input)
    - Better yet: In the minimum distinguishing prefix of the key in the trie.
- (Ignoring `OMap`)
- Better than one-at-a-time insertion into trie.

Generic Tries: Properties

- Tries are based on the laws of exponentials (`Trie K V` $\cong V^K$):

$$V^1 \cong V \qquad V^{K_1+K_2} \cong V^{K_1} \times V^{K_2} \qquad V^{K_1 \times K_2} \cong (V^{K_2})^{K_1}$$

- Correctness:

```
discr o   =  flatten . trie o
sort o    =  concat . discr o
```

  where `flatten :: Trie k v -> List v` flattens a trie into a list by homomorphically interpreting trie constructors as list operations.

- Proofs use strong naturality properties of `discr` and `sort`

Benchmark: Searching the Bible

- Preparatory steps (we use Project Gutenberg's The Bible):

```
bible <- readFile "pg30.txt"
let rel = zip (words bible) [0 ..]
let concordance = build ostring rel
```

- Where is `"God"`?

```
lookup concordance "God"
```

$\Longrightarrow$ `Just [467,496,506,518,527,536,559,583,610,...`

- How frequent is `"God"`?

```
fmap length (lookup concordance "God")
```

$\Longrightarrow$ `Just 2229`

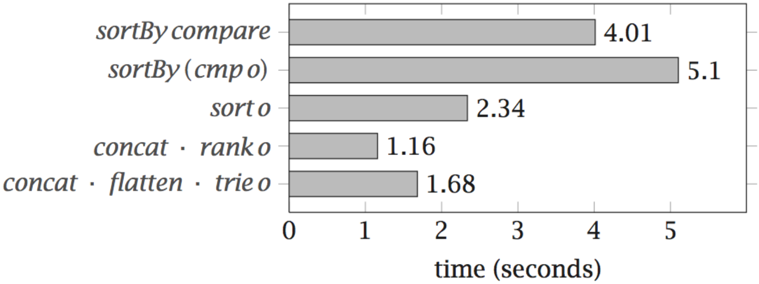
- And the `"devil"`?

```
fmap length (lookup concordance "devil")
```

$\Longrightarrow$ `Just 23`

Benchmark: Performance

Sorting the words of Project Gutenberg's The Bible (5218802 characters, 824337 words).



Summary

- Generic distributive sorting and searching
- Orders are represented syntactically
  - Many sort orders per type, not just standard order
- The separation of keys and values is essential:

```
sort  :: Order k -> List (k, v) -> List v
discr :: Order k -> List (k, v) -> List (List v)
build :: Order k -> List (k, v) -> Trie k (List v)
```

  - Correctness via strong naturality
  - Keys are used *affinely* (used at most once) $\Longrightarrow$ linear time complexity
- Unoptimized Haskell implementation with surprisingly good performance

Related Work

- Cai, J., Paige, R.: *Using multiset discrimination to solve language processing problems without hashing.* Theoretical Computer Science **145**(1-2) (July 1995) 189–228.
- 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), (September 2008) 91–102.
- Henglein, F.: *Generic top-down discrimination for sorting and partitioning in linear time.* Journal of Functional Programming **22**(3) (July 2012) 300–374.
- Connelly, R.H., Morris, F.L.: *A generalization of the trie data structure*. Mathematical Structures in Computer Science **5**(3) (September 1995) 381–418.
- Hinze, R.: *Generalizing generalized tries.* Journal of Functional Programming **10**(4) (2000) 327–351.