Generic discrimination

Fritz Henglein
henglein@diku.dk

Department of Computer Science
University of Copenhagen
(DIKU)

SSGEP, Oxford, 2015-07-06

Some simple problems

- Given a list of pointers, *how many unique elements* does it contain?
- Given a list of pointers, what are its *unique elements*.
- Given a list of pointer lists, what are its *unique pointer lists*.
- Given a list of pointer sets (given as lists considered equal modulo permutations and duplicates), what re its *unique pointer sets*.
- How efficient are your solutions?
- Do they allow copying garbage collection?

Great speed, no abstraction

The C and Java solution:

- Convert pointers to numbers (casting, hashing)
- Use address arithmetic and table lookups.

Consequences:

- Potentially nondeterministic program behavior: `ptr = new ... ; if ptr < 4000 then ...`
- No data abstraction: Cannot change implementation, impedes garbage collection.

Great abstraction, no speed

The ML solution:

- Abstract pointers (references): Allow only equality testing, lookup, allocation, update on pointers.
- Use pairwise comparisons.

Great abstraction, no speed

The ML solution:

- Abstract pointers (references): Allow only equality testing, lookup, assignment, update on pointers.
- Use pairwise comparisons.

Consequence:

Theorem

Determining the number of unique elements in a list of n ML references takes $\Omega(n^2)$ *equality tests.*

Have your cake and eat it too?

Question: Can you sort and partition

- *generically*: user-defined orders/equivalences;
- *fully abstractly*: pairwise comparisons determine output;
- *scalably*: worst-case *linear* time[1]

expressed with very short and simple program code?

Answer: Yes

---

[1] for standard and many other orders/equivalences

Great abstraction, great speed

```
part :: Equiv k -> [k] -> [[k]]

numUniqElems ps = length (part eqRef ps)
uniqElems ps = [ head b | b <- part eqRef ps ]
uniqPtrLists pss = [ head b | b <- part (listE eqRef) pss]
uniqPtrSets pss = [ head b | b <- part (SetE eqRef) pss ]
```

- Worst-case linear time
  $(= O(\text{number of pointer occurrences}))$
- Same order of pointers before and after garbage collection (= *as if* using only equality tests to compute result)

Key ingredient: Discriminator

Provide *n*-ary "comparison" function

```
discRef :: [(Ref a, v)] -> [[v]]
```

to data type instead of binary comparison function

```
== :: Ref a -> Ref a -> Ref a
```

- Fully abstract: Only pairwise equality can be "observed" through `discRef` (like having only ==).
- Asymptotically optimal performance: $O(n)$ worst-case time (like treating pointers—internally—as numbers).

Generic discrimination: Method

- Expressive *domain-specific language* for defining orders and equivalences *compositionally*.
- Inherently efficient (usually linear-time and fully abstract) *discriminators* by structural recursion ("generically") on order and equivalence representations.
- Partitioning, sorted partitioning, sorting, joining functions, etc., as *applications* of discriminators.

Example: Word occurrences

Word occurrences, alphabetically sorted:

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

Word occurrences, in order of occurrence in input

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

Example: Word occurrences, case insensitive

Definition of alphabetic order, but case-insensitive:

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

Word occurrences, case insensitive, alphabetically sorted:

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

Word occurrences, case insensitive, order of occurrence in input:

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

Orders

Definition (Total preorder)

A *total preorder (order)* $(T, \leq)$ is a type T together with a binary relation $\leq \subseteq T \times T$ that is reflexive, transitive and total.

Order constructions

Constructions for defining new orders from old:

- Trivial order on any type
- Standard total orders on primitive types
- Constructions:
  - Lexicographic order (on pair types)
  - Sum order (on sum types)
  - Induced order on domain type by a function to an ordered range type
  - Recursion
  - Inverse order, etc.
- Let's look at some examples.

Order expressions

A *typed language* of order constructions:

```
data Order t  where
     Nat  ::  Int -> Order Int
     Triv ::  Order t
     SumL ::  Order t1 -> Order t2 -> Order (Either t1 t2)
     PairL::  Order t1 -> Order t2 -> Order (t1, t2)
     MapO ::  (t1 -> t2) -> Order t2 -> Order t1
     Inv  ::  Order t -> Order t
     BagO ::  Order t -> Order [t]
     SetO ::  Order t -> Order [t]
```

Implicit recursion is allowed. So order expressions may be infinite.
(Think of them as potentially infinite trees.)
Examples:

```
ordChar8 = MapO ord (Nat 255)
listL r = MapO fromList
         (SumL ordUnit (PairL r (listL r)))
```

Generic definition of comparison functions

```
lte :: Order t -> t -> t -> Bool
```

- Definitional interpreter (= denotational semantics of order representations)
- Idea: Structural recursion on first argument (the order expression)

```
lte (Nat n) x y  = x <= y
lte Triv    x y  = True
...
lte (PairL r1 r2) (x1, x2) (y1, y2) =
     lte r1 x1 y1 &&
     if lte r1 y1 x1 then lte r2 x2 y2 else True
lte (MapO f r) x y = lte r (f x) (f y)
lte (Inv r) x y = lte r y x
```

Generic definition of sorting functions

- Generic definition of `lte` corresponds to *compositional* definition of comparison functions; e.g.
  - **Q**: Given comparison functions `lte r1` and `lte r2`, how to construct a comparison function `lte (PairL r1 r2)` for the product order?
  - **A**:

    ```
    lte (PairL r1 r2) (x1, x2) (y1, y2) =
          lte r1 x1 y1 &&
          if lte r1 y1 x1 then lte r2 x2 y2 else True
    ```
- Sorting using a comparison function entails $\Omega(n \log n)$-lower bound on number of comparisons
- Why not define *sorting functions* generically (by structural recursion on order expressions)?

Generic definition of sorting functions: Problem

```
sort :: Order k -> [k] -> [k]
```

- Imagine now we want to define the case for `Pair r1 r2`:

```
sort (Pair r1 r2) xs =
   ... sort r1 ... sort r2 ...
```

- How to do this?
- We need to sort lists of *pairs*, but both `sort r1` and `sort r2` can only sort lists of single components—association of components is lost.
- Does not work!
- Idea: Allow for "satellite data" to be associated with keys to be sorted.

Discriminator

Definition (Discriminator)

A function Δ is a *discriminator* for equivalence relation E if

- it maps a list of key-value pairs to a list of *groups*, where each group contains the value components that are associated with E-equivalent keys in the input (partitioning property);
- it is parametric in the value components: For all binary relations Q, if $\vec{x}\,(id \times Q)^*\,\vec{y}$ then $\Delta(\vec{x})\;Q^{**}\;\Delta(\vec{y})$ (parametricity property).

Order discriminator

Definition (Order discriminator)

Δ is an *order discriminator* for ordering relation O if it

- is a discriminator for $\equiv_O$, the equivalence relation canonically induced by O, and
- returns the groups of values in ascending O-order on the keys giving rise to them (ordered partitioning property).

Partial abstraction

Definition (Key equivalence)

Let P be an equivalence relation. Lists $\vec{x}$ and $\vec{y}$ are *key equivalent under P* if $\vec{x}\,(P \times id)^*\,\vec{y}$.

Definition (Partially abstract discriminator)

A discriminator Δ for equivalence relation E is *partially abstract* if $\Delta(\vec{x}) = \Delta(\vec{y})$ whenever $\vec{x}$ and $\vec{y}$ are key equivalent under E.

- Result does not depend on particular equivalence class representative.
    - E.g., the particular list representation under set equivalence: $\Delta([([1,4,5],100),([2,3],200)) =$ $\Delta([([5,1,4],100),([3,2],200))$

Full abstraction

Definition (*R*-correspondence)

Let R be an equivalence relation. 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 *R-correspondent*, written $\vec{x} \cong_R \vec{y}$, if $m = n$ and for all $i, j \in \{1 \ldots n\}$ we have $v_i = w_i$ and $k_i \mathrel{R} k_j \Leftrightarrow l_i \mathrel{R} l_j$.

Definition (Fully abstract equivalence discriminator)

A discriminator is a *fully abstract equivalence discriminator* for E if it respects E-correspondent inputs: For all $\vec{x}, \vec{y}$, if $\vec{x} \cong_E \vec{y}$ then $\Delta(\vec{x}) = \Delta(\vec{y})$.

- Result depends only on which pairwise equivalences hold between the input keys.

Full implies partial

Proposition

A fully abstract discriminator is also partially abstract, but not necessarily vice versa.

Example

Let `D` be a fully abstract equivalence discriminator.

- $(x, y) \in E_0$ iff both x, y even or both odd.
- Possible result:
  $\texttt{D}[(\mathbf{5}, 100), (\mathbf{4}, 200), (\mathbf{9}, 300)] = [[100, 300], [200]]$
- By parametricity, then also:
  $\texttt{D}[(\mathbf{5}, "\mathit{foo}"), (\mathbf{4}, "\mathit{bar}"), (\mathbf{9}, "\mathit{baz}")] = [["\mathit{foo}", "\mathit{baz}"], ["\mathit{bar}"]]$
- By partial abstraction, then also:
  $\texttt{D}[(\mathbf{3}, 100), (\mathbf{8}, 200), (\mathbf{1}, 300)] = [[100, 300], [200]]$
- By full abstraction, then also:
  $\texttt{D}[(\mathbf{16}, 100), (\mathbf{29}, 200), (\mathbf{4}, 300)] = [[100, 300], [200]]$

Partitioning and sorting from discrimination

Sorted partitioning from order discrimination:

```
spart :: Order t -> [t] -> [[t]]
spart r xs = sdisc r [ (x, x) | x <- xs ]
```

Sorting from sorted partitioning:

```
dsort :: Order t -> [t] -> [t]
dsort r xs = [ y | ys <- spart r xs, y <- ys ]
```

Unique sorting (no duplicates modulo equivalence) from sorted partitioning:

```
usort :: Order t -> [t] -> [t]
usort r xs = [ head ys | ys <- spart r xs ]
```

Basic order discrimination: Bucket sorting

- `sdisc` requires basic order discriminator `sdiscNat n` for (the standard order on) small integers $[0 \dots n]$.
- Use bucketing:
  1. Allocate/reuse bucket table $T[0 \dots n]$, initialized to empty lists.
  2. For each key-value pair (k, v) in input, add v to $T[k]$.
  3. For $0 \leq i \leq n$ in ascending order,
     if $T[i]$ nonempty, append contents to output and reset $T[i]$ to empty.
- Note: Last step requires n (size of bucket table) steps, even if input is very small.

In Haskell:

```
sdiscNat n xs = filter (not . null) (bucket n update xs)
   where update vs v = v : vs
bucket (n :: Int) update xs =
   reverse (elems (accumArray update [] (0, n) xs))
```

Pair discrimination

```
sdisc (PairL r1 r2) xs   =
   [ vs | ys <- sdisc r1 [ (k1, (k2, v)) | ((k1, k2), v) <- xs ]
          vs <- sdisc r2 ys ]
```

1. Discriminate on first component of keys.
2. For each resulting group, discriminate on second component.

Generic order discrimination

- `sdisc` : A stable generic order discriminator.
- The complete code (except for `BagO`, `SetO`):

```
sdisc :: Order k -> [(k, v)] -> [[v]]
sdisc r []                  = []
sdisc r [(k, v)]            = [[v]]
sdisc (Nat n) xs            = sdiscNat n xs
sdisc Triv xs               = [map snd xs]
sdisc (SumL r1 r2) xs       = sdisc r1 lefts ++ sdisc r2 rights
   where (lefts, rights) = split xs
sdisc (PairL 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 (Inv r) xs            = reverse (sdisc r xs)
```

Asymptotic time complexity

Theorem

For each finite `r` *the function* `sdisc r` *executes in worst-case linear time.*

Proof: Induction on `r`.

Note:

- The linear factor depends on `r`.
- Applies only to nonrecursive types of elements ("finite").

Asymptotic time complexity

Theorem

Let $R \in \mathcal{R}^{\infty}$ *and* $R' \in \mathcal{R}[r_1]$ *such that*

$$R = \mathrm{MapO}\, f\, (R'[R/r_1])$$

where $R :: \mathit{Order}\ T$ *and* $R' :: \mathit{Order}\ T'$*. Furthermore let* $f :: T \rightarrow T'[T/t_1]$ *be such that* $|f(k)| \leq |k|$ *and* $\mathcal{T}_f(k) = O(|f(k)|^{T'})$*.*
Then $R \in \mathcal{L}$*:* R *is linear-time discriminable.*

Corollary

For all standard orders `r` *on first-order regular recursive types,* `sdisc r xs` *executes in linear time.*

Nonlinearity

- Standard lexicographic ordering:

```
listL p = MapO fromList
              (SumL ordUnit (PairL p (listL p)))
```

- Flip-flop ordering on lists: Compare last elements, then first, then next-to-last, then second . . . :

```
flipflop p = MapO (fromList . reverse)
              (SumL ordUnit (PairL p (flipflop p)))
```

Observe:

- `sdisc (listL ordChar8)`: Linear time.
- `sdisc (flipflop ordChar8)`: Quadratic time.

Linear-time: Idea

The recursive type can be polymorphically abstracted in `listL`:

```
listL p = MapO fromList
            (R (listL p))
  where R :: Order t -> Order (Either () (Char, t))
        R r = SumL ordUnit (PairL p r)
```

- Only *parametric polymorphic* functions can occur in abstracted constructor `R`, which cannot “touch” (access) those parts of the input that are passed to the recursive calls of the same discriminator.
- Not possible for `flipflop`— occurrence of `reverse` in abstracted version is not typable.

Basic equivalence discrimination

- Instead of bucket sort, use *basic multiset discrimination* (Cai, Paige 1994).
  - Like bucket sort, but
  - Traverse table in *key insertion order*.
- Yields a fully abstract integer equality discriminator.
- Performance even better than bucket sorting: Final traversal of whole array avoided.
  - No dynamic bucket table allocation required.
  - Use single static bucket array (per thread).

Basic equivalence discriminator in Haskell

```
discNat :: Int -> [(Int, v)] -> [[v]]
discNat size =
  unsafePerformIO (
  do table <- newArray (0, size) [] :: IO (IOArray Int [v])
     let discNat’ xs = unsafePerformIO (
         do ks <- foldM (\keys (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 (\vss k -> do elems <- readArray table k
                                 writeArray table k []
                                 return (reverse elems : vss))
                       [] ks )
     return discNat’ )
```

Generic equivalence discrimination

```
disc :: Equiv k -> [(k, v)] -> [[v]]
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) x        = discColl updateSet e xs
```

Abstraction properties

Theorem (Full abstraction of `sdisc`)

`sdisc` is fully abstract (for ordering) and stable.

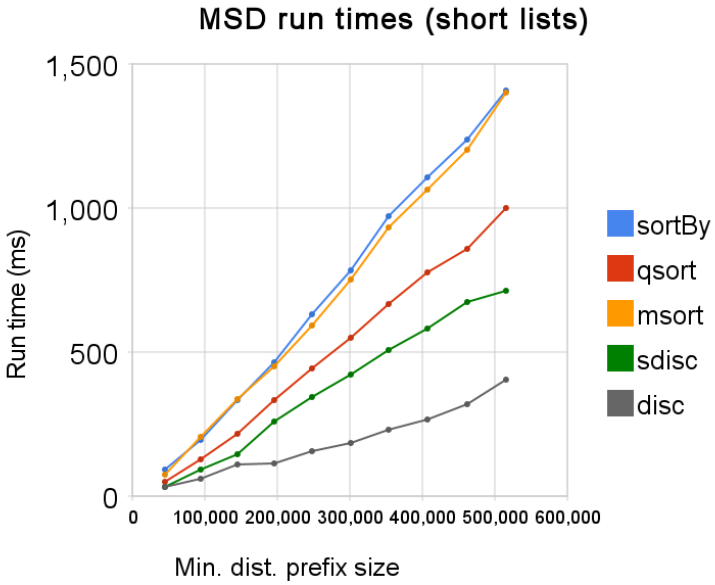
Theorem (Abstraction properties of `disc`)

`disc` is partially abstract (for equivalence) for equivalences not containing `BagE` and `SetE`.

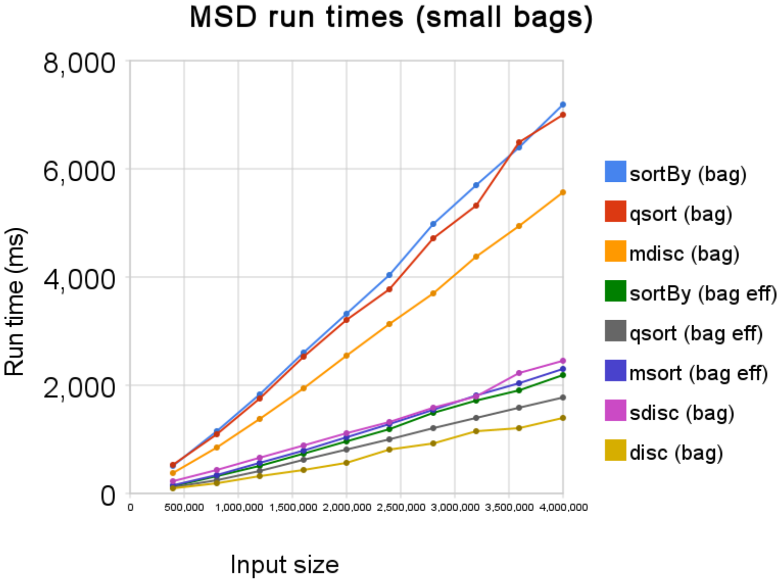
Theorem (Fully abstract equivalence discrimination)

There is a fully abstract generic discriminator `edisc` with the same asymptotic performance as `disc` and `sdisc`.

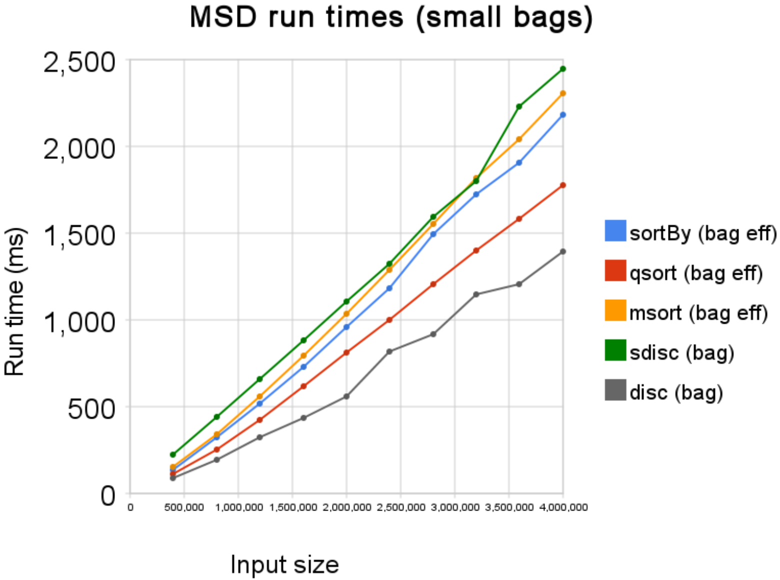
Performance



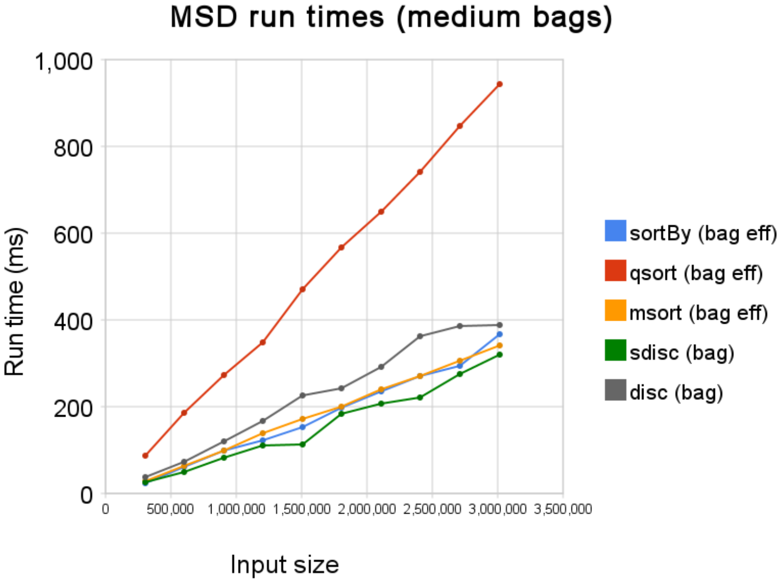
Performance



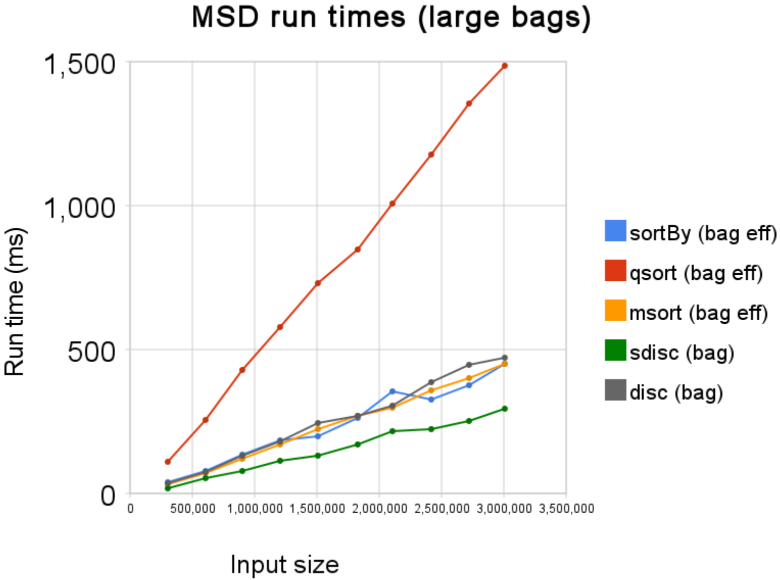
Performance



Performance



Performance



Variations, extensions

- Domain-theoretic semantics
- Avoiding sparse bucket table traversals for order discrimination and sorting
- Equivalence expressions (analogous to order expressions)
- Bag and set equivalence discrimination
- Run-time order and equivalence normalization for correctness and efficiency
- Combinatory discriminator library (without order/equivalence expressions, requires rank-2 polymorphic types)
- Comparison with complexity of sorting
- Generic tries
- Nontrivial applications: AC-term equivalence, type isomorphism, **equijoins**

Select related work

- Paige et al. (1987-97): Basic multiset discrimination for pointers, strings, acyclic graphs; application to lexicographic sorting
- Henglein (2003): Unpublished note on multiset discrimination (top-down, bottom-up) and algorithms for circular data structures
- Ambus (MS thesis, 2004): Java discriminator library, internal and external (disk) data, application to asynchronous data coalescing in P2P-based XML Store (see plan-x.org)
- Henglein (ICFP 2008): Order discriminators
- Henglein (JFP, Nov. 2012): *Generic top-down discrimination for sorting and partitioning in linear time*
- Henglein, Hinze (APLAS 2013): Generic Sorting and Searching
- Kmett (2015): Generic discrimination, streaming

Open problems ("Homework")

- Generic bottom-up discrimination (for acyclic data structures)
- Generic cyclic discrimination (for cyclic data structures)
- Staged implementation (partial evaluation)
- Parallel discrimination

Take-home message: GAS

Simultaneously:

- **Genericity:** DSL for orders and equivalences for correctness and safety/limited expressiveness
- **Abstraction:** Statically guaranteed representation independence
- **Scalability:** Asymptotically optimal computational performance

All equi-abstract interfaces are equivalent, but some are faster than others.

Program

1. Today: Generic discrimination
2. Tomorrow: Generic multiset programming
3. Thursday: Generic linear algebra

End of talk