ORACLE

Classifiers — a new basic API?

theory and practice

John Rose, JVM Architect

May 26, 2022

Classifiers, in the abstract…

a classifier is a fast function from a key set K to an index result

- Works like key -> STATIC_LIST.indexOf(key), mapping M=|K| keys to [-1..N-1]
  - Typically M=N and the original key set has an intrinsic order; sometimes N>M.
  - Also works like key -> (int) STATIC_MAP.get(key) with null mapped to -1
- Must be fast, meaning O(1), so there is an algorithmic requirement
  - (many interesting algorithms for small sets and others for large ones)
- The key set can be drawn from any domain: char, long, string, array, record
  - The key set must be equipped with some hash functions, at least one
  - (spoiler: so you can go from keys to sparse ints, and then to dense ints)
- The result set might as well be a compact set of integers
  - You can get other kinds of results in O(1) using an extra table lookup

internal building block for classifiers: weak classifiers

- (My term; they are called “static functions” in the literature.)
- Also a classifier on some original set K of M keys, *but with relaxed requirements*
- **Non-exclusive:** Can return **any value** when presented with a non-key k∉K
- Non-dense: Can return ints in an **expanded range** [0..N], where N ≈ M·(1+ε)
- Permuted: The indexes it returns are **chosen by it**, not the key set
  - That is, WC(k1)≠WC(k2) are different values if (a) k1≠k2, and (b) k1,k2∈K
- Must still be O(1). For small M, can be a few machine instructions.

- (This is where [semi-]perfect [almost-]minimal hash functions are useful.)

lots of literature to draw on for framing the problem and engineering it

- “Broadword Implementation of Rank/Select Queries” (Vigna 2020)
- “RecSplit: Minimal Perfect Hashing via Recursive Splitting” (Esposito/Graf/Vigna 2019)
- “Engineering Compressed **Static Functions**” (Genuzio/Vigna 2018)
- “Retrieval and Perfect Hashing using Fingerprinting” (Muller/… 2014)
- “Hash, Displace, and Compress” (Belazzougui/Botelho/Dietzfelbinger 2009)
- “Linear Hash Functions” (Alon/Dietzfelbinger/… 1999)
- “A family of perfect hashing methods” (Majewski/… 1996)
- “A representation for multinomial cumulative distribution functions “ (Levin 1981)

[post-talk slide] a few more points (only touched on verbally)

- other possible names: indexer, categorizer, recognizer (for REs and parsing)

- a classifier is like a list, but with only “one good operation”, indexOf
- it is almost a functional interface, except that it also reports its range limit N
- it does *not* report, and may not even record, the keys from which it was built

- one might say that a classifier which classifies a finite key set is “discrete”
- while a classifier that can classify an unlimited number of keys is “continuous”
- (the “continuous” case arises when using classifiers for pattern switches)

Switching the subject…

compilation of an int switch statement in Java

switch ((int)i) {
case 1: … case 42: … }

⇒

iload #i
lookupswitch {
1: …
42: …
}

compilation of an enum switch statement in Java

```
enum RGB { R, G, B }
switch ((RGB)e) {
case G:  gstuff(); …  case B: bstuff(); …   }
⇒

static {  // set up lazily on first use (cf. lazy resolution)
  $map = new int[RGB.values().length];
  $map[G.ordinal()] = 1;    $map[B.ordinal()] = 2;  }
switch ($map[e.ordinal()]) {
case 1: gstuff(); …  case 2: bstuff(); … }
⇒ tableswitch { case 1: … case 2: … }
```

compilation of a string switch statement in Java

switch ((String)s) {
case "foo": foostuff(); … case “bar": barstuff(); … }

⇒

switch (]) {
case #(foo.hash): foostuff(); … case #(bar.hash): barstuff(); … }

⇒

apush #s; invokevirtual Object::hashCode
lookupswitch {
#(foo.hash): if (s.equals(“foo")) { foostuff(); … } …

existing tactics

- dense int set: lookupswitch, probably a jump table
- sparse int set: tableswitch; combo of jump tables & binary search
- enum set: spin up a permutation table to map dense to dense ints
- string set: map sparse hash keys to cases
  - validate each string with Object::equals after its hash matches
  - use cascading validation (linear search) in case of hash collision
- pattern-switch: do something complicated with condy & runtime support

potential tactics for int switch

- compress sparse to dense keys using binary search
  - this is the data-oriented alternative to if/else binary search
- compress sparse to dense keys using perfect hash
  - faster than any kind of binary search, on a good day
  - easier for smaller numbers of keys; large keys sets are hard
- fall back to binary search or (large key sets) a regular hash table

- if the switch just produces a value per case, use real table lookup
- cook any required tables down into read-only C data (Panama)
  - *Java arrays are bad for this: the JIT must treat them as non-constant*

potential tactics for enum switch

- same as for int switch: enum ordinals are a nice dense key set
- but it cannot be done at javac time: *Enum::ordinal is unpredictable*
  - (compare with string switch, where String::hash is predictable)

- solution: use indy/condy to set up a classifier after the ordinals are set in stone

- benefit: JIT can “see through” suitably prepared read-only classifier tables
- benefit: less classfile bulk (javac should not be arranging classifier logic)

potential tactics for string switch

- use String::hashCode when there are no collisions
  - use a different one (randomly chosen) to avoid collisions if necessary
- should not be done at javac time

- solution: use indy/condy to set up a classifier

- benefit: JIT can “see through” suitably prepared read-only classifier tables
- benefit: less classfile bulk (javac should not be arranging classifier logic)
- benefit: no cascading internal switches for the JIT to work through

[post-talk slide] potential tactics for pattern switch

- each pattern (and sub-pattern) can be recognized by a classifier
  - example composition: /case Foo f:/ can use getClass and classify the mirror
- this classifier is usually “continuous” not “discrete” (puts many items in one class)
- but if pattern bindings exist, something richer is needed
  - start with a classifier, but have it return an optional binding bundle
  - the bundle is of a different (“dependent”) type for each result index ∈ [-1..N-1]
  - it can be envisioned as an enriched subtype of Classifier<T>
- again, indy/condy and runtime APIs can factor out unwanted “classfile goo”
- it is obviously best that such APIs make sense by themselves, for separate use
  - and, as Stuart pointed out, they must be secure even if fed surprise inputs

More concretely…

public interface Classifier<T>

Represents a function that accepts one argument {@code x} and returns a classification index in a limited range.
<p>
The following example illustrates a switch statement built on a string classifier that emulates a native Java string switch:
<pre>{@code
    static final Classifier<String> CFR
        = Classifier.rejectingNull(
            Classifier.ofStrings("zero", "one", "two", "three"));
    …
    return switch (CFR.classify(x)) {
      case 0 -> "it was zero";
      case 1 -> "it was one";
      case 2 -> "it was two";
      case 3 -> "it was three";
      default -> "not null, but none of the above";
    };
}</pre>

public interface Classifier<T>

Each classification index is either {@code -1} meaning the classifier refused to assign a classification, or else it is a non-negative integer value {@code i < n}, where the exclusive limit {@code n} is the number of classifications supported by the classifier.

Thus, a classifier acts as a integer-valued function with an advertised range, of {@code [-1,n-1]}. The size of the range may be queried by the {@link #numberOfClassifications()} method.

public interface Classifier<T>

A classifier must always assign equal arguments equal classifications, in particular assigning the same object reference or same primitive value or same string a constant classification index across multiple classification requests.

Classifiers are likely (though not required) to query the methods {@link Object#equals}, {@link Object#hashCode}, and (if available) {@link Comparable#compareTo} of their inputs.

<p>

Unless it is specifically advertised to reject the some inputs (such as the {@code null} reference), the {@link #classify()} method of a classifier must return normally for all possible inputs. It may return a valid classification index, or if it refuses to classify an input, it must return the sentinel value {@link #UNCLASSIFIED}.

public interface Classifier<T>

These conventions are coherent with those of {@link List#indexOf}, as if a list of items were interpreted as specifying that each list element is assigned the classification index of its position in the list. (Any subsequent duplicates are ignored and their positions are never reported as successful classifications.) The sentinel value {@code -1} reported for elements not contained in the list is identical with the value {@link #UNCLASSIFIED} used here. Of course, non-trivial classifiers are expected to use better algorithms than linear search.

a few design issues

Should classifier extend `ToIntFunction`, or is it just fine to say `cfr::classify` to get a function? (Default answer: Leave out the extend relation.) A classifier is really just an int-function plus an enforced range with a query about that range. But it is also something higher-level than a mere function.

Similarly, should an unboxing classifier (`Classifier.Unboxing`) extend the generic classifier API? This seems useful given the interesting range of possible classifier combinators, but it does entail boxing along some paths. Note that `IntStream` does not extend `Stream<Integer>`.

Should there be separate package members for `IntClassifier`, etc.?

On the other hand, should we have just `LongClassifier` and call it a day? You can classify any primitive value by encoding it as a long and then classifying the long as a bit pattern. Having just one unboxing classifier type (and one boxed one) would be simpler.