# Lists

J2H has the abstract class HList<T> to represent Haskell lists. We considered several ways to work with a Haskell List in Java, when using numbers. For instance, if we consider a Long (Haskell Int in 64-bit platforms):

* HList<Long> (**preferred**) – A list of plain Java Longs. This is the most compatible option Java-wise, as HList<T> implements Java List<T>, and it also has the lowest overhead. On the other hand, it does not implement Haskell’s number hierarchy.
* HList<Hint> – A list of Haskell Ints. It provides access to instance functions that are available for Haskell Ints. It can have additional overhead, since the HInt encapsulates a Java Number (i.e., Long).
* HListInt (**preferred**) – A specialized HList of Hints. Can implement numeric instance methods in a better way (e.g., sum() that work for empty lists, helper methods that accept Java types such as Number).
* ListInt – A specialized HList of Longs. The same as HListInt, but with Java Longs and lower overhead. However, it seems to be slightly slower than HList<Long>.

If we want performance, we should use HList<T>, in particular for Java types (e.g., HList<Long>). For Haskell types, we prefer specialized lists (e.g., HListInt).

After some recent performance tweaks, it seems HListInt currently has performance close to HList<T>, however we need to test this with a proper framework (e.g., JMH) before drawing conclusions.

# Functions over Lists

In Java, due to type-erasure, generics do not count towards method signature, so having two methods with the same name and the same generic class is not possible:

Lists.sum(List<Integer> list)

Lists.sum(List<Double> list) (compilation error if both methods are part of a class)

We considered three solutions:

|  |  |
| --- | --- |
| **Solution** | **Examples** |
| Class per list type | IntLists.sum()  DoubleLists.sum()  HIntLists.sum() |
| Change method names | Lists.sumi()  Lists.sumd()  Lists.sumhi() |
| Class per function - **preferred** | Sum.jint()  Sum.jdouble()  Sum.hint() |

We choose to use a class per function. Note, however, that when using specialized lists (e.g., HListInt, which is equivalent to HList<Hint>), we have access to instance methods such as .sum(), which in the general case can be implemented as a call to Sum class. E.g.: abstract class HListNumber, from where classes such as HListInt are derived, has a default implementation of instance method .sum() that call Sum.real(). Particular implementations of HListNumber can override this function to provide more performant versions.

# Performance Results

* Preliminary results indicated that BigInteger vs HInteger seem to be roughly equivalent performance wise

# Static Methods

There are methods that are related to a type, but that are “static” in a Java sense, i.e. they are not called using an already existing instance (e.g., fromNumber() methods, which act as constructors). However, we do not want to implement them as static functions, since that prevents using them in interfaces (e.g., method min/maxBound, in interface Bounded), and also makes them more cumbersome to use.

All Haskell methods that could be seen as “static” methods in the Java sense, in the current implementation are instance methods due to 1) emulating Haskell class hierarchy through interfaces and 2) to implement a fluent API.

The current solution puts these “static” methods on clearly marked interfaces (e.g., HIntStatic for the static methods of Hint), and each implementation type realizes this interface and provides a static field that gives access to these methods, e.g.: Hint.STATIC, which returns a HIntStatic instance. This way it is possible to call fromNumber() without explicitly create an instance. E.g.: HInt.STATIC.fromNumber(10). Note that HInt still extends from HIntStatic, so all its methods are still available from HInt.