Sort.java

```
// FILE. . . . . /home/hak/hlt/src/hlt/osf/base/Sort.java
// EDIT BY . . . Hassan Ait-Kaci
// ON MACHINE. . Hak-Laptop
// STARTED ON. . Mon Sep 02 09:16:33 2013
```

Author: Hassan Aït-Kaci
Version: Last modified on Thu May 09 05:18:36 2019 by hak

```
package hlt.osf.base;

import java.util.HashSet;

import hlt.osf.util.Custom;
import hlt.osf.util.BitCode;
import hlt.osf.util.Decoded;
import hlt.osf.exec.Context;
import hlt.osf.exec.Taxonomy;
import hlt.osf.exec.LockedCodeArrayException;

import hlt.language.util.Comparable;
```

This is the class of an atomic sort's symbol name as explained in the specification.

```
// public class Sort extends SortExpression implements Comparable
public class Sort implements Comparable
  {
```

Evaluation context for this sort as a sort expression.

```
    private Context _context;
```

Returns this sort evaluation context for this sort as a sort expression.

```
    public Context context ()
    {
      return _context;
```

```
}
```

Sets this sort's sort expression encoding evaluation context to the given `hlt.osf.exec.Context`.

```
public Sort setContext (Context context)
{
  _context = context;
  return this;
}

/* ************************************************************************ */
```

SORT EXPRESSION TYPE (a symbol)

```
/* ************************************************************************ */

public byte type ()
{
  return _context.SYM();
}

/* ************************************************************************ */
```

NEW FIELDS, GETTERS, AND SETTERS (other than inherited from `Sort Expression`)

```
/* ************************************************************************ */
```

The private index of this `Sort`.

```
private int _index;
```

Gets this `Sort`'s index.

Returns: the index of this `Sort` in its taxonomy array

```
public int index ()
{
  return _index;
```

```
}
```

Sets the index of this `Sort` to the specified argument (an `int`).

Returns: this `Sort`

```
public Sort setIndex (int index)
{
  _index = index;
  return this;
}

/* ********************************************************************** */
```

The private name of this `Sort`.

```
private String _name;
```

Gets this `Sort`'s name.

Returns: the name of this `Sort` in its taxonomy array.

```
public String name ()
{
  return _name;
}
```

Sets the name of this `Sort` to the specified argument (a `String`).

Returns: this `Sort`

```
public Sort setName (String name)
{
  _name = name.intern();
  return this;
}

/* ********************************************************************** */
```

The private binary code of this `Sort`.

```
private BitCode _bitcode;
```

Gets this `Sort`'s binary code.

Returns: the binary code of this `Sort` in its taxonomy array.

```
public BitCode bitcode ()
{
  return _bitcode;
}
```

Sets this `Sort`'s binary code to the specified `BitCode`.

Returns: this `Sort`.

```
public Sort setBitcode (BitCode code)
{
  _bitcode = code;
  return this;
}

/* ********************************************************************** */
```

CONSTRUCTORS

```
/* ********************************************************************** */
```

Constructs a `Sort` object with the specified name.

```
public Sort (String name)
{
  setName(name);
}
```

Constructs a `Sort` object with the specified index and name.

```
public Sort (int index, String name)
{
  setIndex(index);
  setName(name);
}
```

Constructs a `Sort` object with the specified index, name, and bitcode.

```
public Sort (int index, String name, BitCode code)
{
  setIndex(index);
  setName(name);
  setBitcode(code);
}
```

Constructs a `Sort` object with the specified name and context.

```
public Sort (String name, Context context)
{
  setName(name);
  setContext(context);
}

/* ********************************************************************** */
```

SORT COMPARISON METHODS

N.B.:

- all are delegated to their associated binary codes in the same way as for sort expressions;
- all test whether `code` is `null` and throw a `LockedCodeArrayException` — may be unnecessary and a source of inefficiency for large taxonomies.

```
/* ********************************************************************** */
```

Returns true iff this sort is a subsort of the specified sort.

```
public boolean isSubsortOf (Sort sort) throws LockedCodeArrayException
{
  if (_bitcode == null)
    throw new LockedCodeArrayException("Cannot compare a non-encoded sort");

  return _bitcode.isContainedIn(sort.bitcode());
}
```

Returns true iff this sort is a strict subsort of the specified sort.

```
public boolean isStrictSubsortOf (Sort sort) throws LockedCodeArrayException
{
  if (_bitcode == null)
    throw new LockedCodeArrayException("Cannot compare a non-encoded sort");

  return _bitcode.isStrictlyContainedIn(sort.bitcode());
}
```

Returns true iff this sort is related to the specified sort.

```
public boolean isRelatedTo (Sort sort) throws LockedCodeArrayException
{
  if (_bitcode == null)
    throw new LockedCodeArrayException("Cannot compare a non-encoded sort");

  return _bitcode.isRelatedTo(sort.bitcode());
}
```

Returns true iff this sort is unrelated to the specified sort.

```
public boolean isUnrelatedTo (Sort sort) throws LockedCodeArrayException
{
  if (_bitcode == null)
    throw new LockedCodeArrayException("Cannot compare a non-encoded sort");

  return _bitcode.isUnrelatedTo(sort.bitcode());
}

/* ************************************************************************ */
```

UNIQUE ASSOCIATED `Decoded` OBJECT

```
/* ************************************************************************ */
```

The `Decoded` object corresponding to this sort.

```
private Decoded _decoded;
```

This is a unique `hlt.osf.util.Decoded` object that is systematically associated to every sort. Conversely, the `sort()` method of the `Decoded` object returned by a `Sort` object's `decoded()` method returns this `Sort` object.

Returns: the `Decoded` object corresponding to this `Sort`

```
public Decoded decoded () // throws LockedCodeArrayException
{
  // if (_decoded == null)
  //        throw new LockedCodeArrayException("Attempt to access a sort's non-initialized Decoded form");

  return _decoded;
}
```

Sets this sort's `Decoded` information.

Parameters: `decoded` - a `Decoded` object

Returns: the argument `Decoded` object

```
public Sort setDecoded (Decoded decoded) throws LockedCodeArrayException
{
  if (decoded == null)
    throw new LockedCodeArrayException("Attempt to set a sort's Decoded form to null");

  _decoded = decoded;

  return this;
}

/* ********************************************************************** */
```

This defines the sort code size for all Boolean computations on, and printing of, sort codes. This is necessary for consistency since only bits within this code size do matter. It is set in the `Taxonomy` class upon locking the defined sorts' code array. This constant is taken into account in Boolean operations (essentially by '*not*') and code printing methods defined the `BitCode` class. For all sort codes, all bits of index greater than or equal to this constant are always false. This is also the index of top when it is initialized and installed in the sort code array upon locking it.

```
private static int _BITCODESIZE;
```

Returns the sort-code's size for an encoded taxonomy. Raises a LockedCodeArrayException if the sort code array is not locked.

```
public static int codeSize () throws LockedCodeArrayException
{
  if (!Taxonomy.isLocked())
    throw new LockedCodeArrayException("Attempt to access code size for a non-encoded taxonomy");

  return _BITCODESIZE;
}
```

Sets the sort code size for an encoded taxonomy. Raises a LockedCodeArrayException if the sort code array is locked.

```
public static void setCodeSize (int size) throws LockedCodeArrayException
{
  if (Taxonomy.isLocked())
    throw new LockedCodeArrayException("Cannot change the sort code size for an encoded taxonomy");

  _BITCODESIZE = size;
}

/* ********************************************************************** */
```

This defines top as a static constant. Note that the index is set to 0. But this is temporary, as it will updated to its final index when initialized and be stored in the sort code array. Its final index will then be the value of _BITCODESIZE.

```
private static Sort _TOP = new Sort(0,Custom.TOP_STRING,new BitCode().lock());
```

Returns the top sort as a static constant. This is safe only if the taxonomy this is relative to has been encoded.

```
public static Sort top ()
{
  return _TOP;
}
```

Return true iff this sort is top.

```
public boolean isTop ()
{
  return this == _TOP;
}

/* ********************************************************************** */
```

This defines bottom as a static constant. Note that the index is set to -1. This is because it is not stored in the sort code array and therefore, its index is irrelevant. It is the only sort not stored in the code array and whose height is always 0.

```
private static Sort _BOTTOM = new Sort(-1,Custom.BOT_STRING,new BitCode().lock()).setHeight(0);
```

Returns the bottom sort as a static constant.

```
public static Sort bottom ()
{
  return _BOTTOM;
}
```

Return true iff this sort is bottom.

```
public boolean isBottom ()
{
  return this == _BOTTOM;
}

/* ********************************************************************** */
```

This contains the sorts that are declared as immediate parents of this sort.

```
private HashSet _parents = new HashSet();
```

Returns the declared parents of this sort.

```
public HashSet parents ()
{
  return _parents;
}
```

Adds the specified sort as a parent of this sort, and returns true iff that sort was not already a parent.

```
public boolean addParent (Sort s)
{
  return _parents.add(s);
}
```

Removes the specified sort as a parent of this sort, and returns true iff that sort was indeed a parent.

```
public boolean removeParent (Sort s)
{
  return _parents.remove(s);
}

/* ********************************************************************** */
```

This contains the sorts that are declared as immediate children of this sort.

```
private HashSet _children = new HashSet();
```

Returns the declared children of this sort.

```
public HashSet children ()
{
  return _children;
}
```

Adds the specified sort as a child of this sort, and returns true iff that sort was not already a child.

```
public boolean addChild (Sort s)
{
  return _children.add(s);
}
```

Removes the specified sort as a child of this sort, and returns true iff that sort was indeed a child.

```
public boolean removeChild (Sort s)
{
  return _children.remove(s);
}

/* ********************************************************************** */

boolean minimal = true;
boolean maximal = true;
```

Adds the specified sort to the set of parents of this sort, and this sort to the set of children of the specified sort. Returns true if neither was there before (i.e., already declared to be so). This also maintains the correct set for the parents of bottom (i.e., the minimal sorts) and the children of top (i.e., the maximal sorts).

```
public boolean addIsaDeclaration (Sort sort)
{
  boolean b1 = addParent(sort);
  boolean b2 = sort.addChild(this);

  if (minimal)
    {
      _BOTTOM.addParent(this);
      this.addChild(_BOTTOM);
    }

  if (maximal)
    {
      maximal = false;
      _TOP.removeChild(this);
      this.removeParent(_TOP);
    }

  if (sort.maximal)
    {
      _TOP.addChild(sort);
      sort.addParent(_TOP);
    }

  if (sort.minimal)
    {
      sort.minimal = false;
      _BOTTOM.removeParent(sort);
      sort.removeChild(_BOTTOM);
    }

  return b1 && b2;
}

/* ************************************************************************ */
```

The height of this Sort in its taxonomy. (See the specification.)

```
private int _height = -1;
```

Returns the height of this Sort in its taxonomy. (See the specification.)

```
public int height () throws LockedCodeArrayException
{
  if (!Taxonomy.isLocked())
    throw new LockedCodeArrayException("Can't compute sort heights in a non-encoded taxonomy");

  if (_height != -1)
    return _height;

  return _height = _context.taxonomy().computeHeight(this);
```

```
}

public Sort setHeight (int height)
{
  _height = height;
  return this;
}

public void resetHeight ()
{
  _height = -1;
}

/* ********************************************************************** */
```

Returns the number of subsorts of this sort (not including itself).

```
public int numberOfDescendants ()
{
  return _bitcode.cardinality()-1;
}

/* ********************************************************************** */
```

Returns the set of sorts that are descendants of this sort as a `HashSet`.

```
public HashSet descendants (Taxonomy taxonomy) throws LockedCodeArrayException
{
  return decoded().descendants(taxonomy);
}

/* ********************************************************************** */

// /**
//  * Returns the number of strict supersorts of this sort.
//  */
// int numberOfAncestors (Sort sort)
// {
//   return ancestors().size();
// }
```

Returns the set of sorts that are ancestors of this sort as a `HashSet`.

```
public HashSet ancestors (Taxonomy taxonomy) throws LockedCodeArrayException
{
  //      System.err.println(">>> Decoded = "+decoded());
  return decoded().ancestors(taxonomy);
}

/* ********************************************************************** */
```

BOOKKEEPING UTILITIES

```
/* ************************************************************************ */
```

Returns true iff this is a strict lower sort of the specified sort. This is used by the `hlt.language.tools.Misc.sort` method used in class `Taxonomy` to sort the declared sorts.

```
public boolean lessThan (Comparable sort)
{
  return precedes((Sort)sort);
}
```

This defines the "precedes" ordering as described in the specification. This is a topological ordering that respects the is-a ordering to ease the detection of potential cycles. A cycle is a set of sorts with equal codes after transitive closure has been performed. Using this "precedes" comparison to reorder the array will make all elements in a cycle be contiguous.

```
public boolean precedes (Sort sort)
{
  if (_bitcode.isStrictlyContainedIn(sort.bitcode()))       // this is a proper subsort of sort
    return true;

  if (_bitcode.equals(sort.bitcode()))      // respect the index ordering for equal codes
    return _index < sort.index();

  if (!sort.bitcode().isContainedIn(_bitcode))      // for unrelated sorts
    {
      int thisCodeSize = _bitcode.cardinality();
      int sortCodeSize = sort.bitcode().cardinality();

      if (thisCodeSize == sortCodeSize)
        { // So here, the two codes have same cardinality: the "least"
          // code is the one with the least differring true bit
          int i = _bitcode.nextSetBit(0);
          int j = sort.bitcode().nextSetBit(0);
          while (i == j)
            {
              i = _bitcode.nextSetBit(i+1);
              j = sort.bitcode().nextSetBit(j+1);
            }
          return (i < j);
        }
      else
        return (thisCodeSize < sortCodeSize);
    }

  return false;
}
```

Returns true iff this sort's name is equal to the specified one's.

```
public boolean equals (Sort sort)
{
  return _name == sort.name();
}
```

Locks this sort's bit code - which means that its code will not be modified by the 3 Boolean static bit code operations 'not', 'and', and 'or'.

```
public Sort lock ()
{
  _bitcode.lock();
  return this;
}
```

Unlocks this sort's bit code - which means that its code may be modified by the 3 Boolean static bit code operations 'not', 'and', and 'or'.

```
public Sort unlock ()
{
  _bitcode.unlock();
  return this;
}
```

Returns true iff this sort's bit code is locked - which means that its bit code will not be modified by the 3 Boolean static operations 'not', 'and', and 'or'.

```
public boolean isLocked ()
{
  return _bitcode.isLocked();
}
```

Returns a string form of this Sort object. This string is its name.

```
public String toString ()
{
  return _name;
}
}
```

This file was generated on Thu Jul 11 07:01:29 PDT 2019 from file `Sort.java`
by the `hlt.language.tools.Hilite` *Java tool written by Hassan Aït-Kaci*
