Taxonomy.java

```
// FILE. . . . . /home/hak/hlt/src/hlt/osf/exec/Taxonomy.java
// EDIT BY . . . Hassan Ait-Kaci
// ON MACHINE. . Hak-Laptop
// STARTED ON. . Mon Sep 02 15:35:28 2013
```

Author: Hassan Aït-Kaci
Version: Last modified on Thu Jul 11 14:48:48 2019 by hak

```
package hlt.osf.exec;

import java.util.HashSet;
import java.util.Iterator;

import java.io.PrintStream;
import java.io.IOException;
import java.io.FileNotFoundException;

import hlt.osf.io.TaxonomyLoader;
import hlt.osf.io.BadTokenException;

import hlt.osf.base.Sort;
import hlt.osf.io.DisplayManager;
import hlt.osf.util.Custom;
import hlt.osf.util.BitCode;
import hlt.osf.util.Decoded;
import hlt.osf.util.LockedBitCodeException;

import hlt.language.tools.Misc;
import hlt.language.tools.Debug;
import hlt.language.util.ArrayList;
import hlt.language.util.IntArrayList;
import hlt.language.util.IntIterator;
```

This class defines all the necessary methods for the representation and manipulation of a sort taxonomy. It extends `ArrayList` for storing `hlt.osf.base.Sort` objects. It has built-in sorts and more sorts can be added to it by declaring sorts as subsorts of other sorts. When all the sorts of a taxonomy have been declared, they can then be compiled into `hlt.osf.util.BitCode`s at which point the taxonomy is locked and no more sorts may be added declared into it unless unlocked. An encoded taxonomy can also be saved to, and restored from, disk.

```
public class Taxonomy extends ArrayList implements Contextable
  {
    /* ************************************************************************ */
```

Construct a taxonomy.

```
public Taxonomy ()
{
  super();
}
```

Construct a taxonomy of the specified initial capacity.

```
public Taxonomy (int initialCapacity)
{
  super(initialCapacity);
}
```

Construct a taxonomy with the specified Context.

```
public Taxonomy (Context context)
{
  this();
  _context = context;
}
```

Construct a taxonomy of the specified initial capacity and the specified Context.

```
public Taxonomy (int initialCapacity, Context context)
{
  this(initialCapacity);
  _context = context;
}

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

This taxonomy's operational context.

```
private Context _context;
```

Returns this taxonomy's operational context.

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

Sets this taxonomy's execution context to the specified one, and registers the built-in sorts for this context into this taxonomy; returns this taxonomy.

```
public Taxonomy setContext (Context context)
{
  _context = context;

  for (int i=0; i<builtins.size(); i++)
    {
      Sort sort = (Sort)builtins.get(i);
      add(sort);
      sort.setContext(_context);
      _context.namedSorts().put(sort.name(),sort);
      _context.codeCache().put(sort.bitcode(),sort.setDecoded(new Decoded(sort)).decoded());
    }

  return this;
}

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

This returns the sort with specified index in this taxonomy.

```
public Sort getSort (int index)
{
  return (Sort)get(index);
}

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

This is false as long as this sort taxonomy has not been encoded; true otherwise.

```
private static boolean _isLocked = false;
```

Returns true iff this sort taxonomy has been been encoded.

```
public static boolean isLocked ()
{
  return _isLocked;
}
```

Locks this taxonomy. This is *private* because it should not be accessible from outside this `Taxonomy`.

```
private void _lock ()
{
  _isLocked = true;
}
```

Unlocks this taxonomy. This is *public* because it may be accessible from outside this `Taxonomy`.

```
public void unlock ()
{
  _isLocked = false;
}

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

The first elements of a taxonomy are used to accommodate built-in sorts populated by constants such as integers, floating-point numbers, *etc.*,... Built-in sorts are also stored in an array list called `builtins`. They are actually stored in the initial slots of this taxonomy by the `setContext` method when setting the context of this taxonomy.

```
static public ArrayList builtins = new ArrayList();
```

This creates and returns a new built-in sort with the specified name, adding it to this taxonomy.

```
static private Sort _newBuiltInSort (String name)
{
  // create a new code for the new built-in sort:
  BitCode code = new BitCode();
  // set to 1 the new built-in sort's code's bit corresponding to
  // the current number of built-ins:
  code.set(builtins.size());
  // create a new built-in sort with an index equal to the current
  // number of built-ins, the specified name, and the new code:
  Sort sort = new Sort(builtins.size(),name,code);
  // install the new built-in sort as a child of top:
  Sort.top().addChild(sort);
  sort.addParent(Sort.top());
  // install the new built-in sort as a parent of bottom:
  Sort.bottom().addParent(sort);
  sort.addChild(Sort.bottom());
  // add the new built-in sort to the built-ins:
  builtins.add(sort);
  // lock and return the newly created built-in sort:
  return sort.lock();
}
```

The following static declarations define the built-in sorts.

```
static public final Sort INTEGER = _newBuiltInSort(Custom.INT);
static public final Sort CHAR    = _newBuiltInSort(Custom.CHAR);
static public final Sort FLOAT   = _newBuiltInSort(Custom.FLOAT);
static public final Sort STRING  = _newBuiltInSort(Custom.STRING);
static public final Sort BOOLEAN = _newBuiltInSort(Custom.BOOLEAN);

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

Encodes all the sorts of this taxonomy and initializes it.

```
public void encodeSorts () throws AnomalousSituationException
{
  // encode all the non-built-in defined sorts in this sort code array:
  encodeSorts(builtins.size(),size()-1);
  // initialize this taxonomy's sorts in the context and caches:
  _initialize();
}
```

Encodes the sorts between index first and index last (both inclusive) of this sort code array and check for cycles, committing all defined sorts in the specified range.

It encodes the taxonomy performing transitive closure using bottom-up encoding, which is much more efficient than the general complete $O(n^3)$ method: it is simply $O(n)$ as it sweeps the sort array just once encoding each sort as the `boolean` '*or*' of its immediate subsorts (children) in a layer starting from one containing only bottom. The next layer is obtained as the union of all the parents of the current layer minus the sorts that do not have all their children encoded yet (which means that they can be reached later). See more details in CEDAR Technical Report No.2 and CEDAR Technical Report No.4.

However, this method is garanteed to compute the transitive closure only for acyclic posets. Indeed, computing a layer may not be done correctly if there are loops in the taxonomy. This means that after proceeding with a bottom-up encoding, if there are sorts that are still unlocked, then this implies that there are cycles.

While the existence of cycles may be detected (by a simple check that all codes are locked), it is non trivial to identify and enumerate all the cycles. But cycle identification is much easier if encoding was done with the general transitive closure method in $O(n^3)$.

In order to do so, we can restrict the taxonomy only to unlocked codes (since they identify all sorts that belong to cycles), and then re-encode them using the $O(n^3)$ method. This is viable since in general the number of still unlocked codes is a relatively small subset of the whole taxonomy.

```
public void encodeSorts (int first, int last)
  throws AnomalousSituationException, LockedCodeArrayException, CyclicSortOrderingException
{
  if (first > last)
    throw new AnomalousSituationException("No sorts have been defined to encode");

  if (_isLocked)
    throw new LockedCodeArrayException("Unable to encode an already encoded sort taxonomy");

  // perform the encoding by transitive closure:
  System.out.println("*** Performing transitive closure...");
  long time = System.currentTimeMillis();
  // _transitiveClosure(first,last);        // this is too expensive - O(n^3)
  _bottomUpClosure();                       // this is linear - but see comment below
  time = System.currentTimeMillis() - time;
  System.out.println("*** Transitive closure processing time = "+time+" ms");

  // Verify that all sorts have been encoded correctly:
  System.out.println("*** Performing consistency check of the taxonomy...");
  time = System.currentTimeMillis();
  _checkCodes(first,last);
  time = System.currentTimeMillis() - time;
  System.out.println("*** Code consistency check processing time = "+time+" ms");

  // Commit all defined sorts:
  System.out.println("*** Committing all sorts...");
  time = System.currentTimeMillis();
  _commitAllSorts(first,last);
  time = System.currentTimeMillis() - time;
  System.out.println("*** Sort commitment processing time = "+time+" ms");
```

```
}
```

Locks this taxonomy, initializes the top sort to have the correct number of bits set, sets the top and bottom sorts, as well as the built-in sorts to have the right context and registers them into the named sort array and the code cache.

```
private void _initialize () throws AnomalousSituationException
{
  // sets the sort code size to be the size of this code array + 1
  // to account for top (added further down in this method):
  Sort.setCodeSize(size()+1);

  // lock this code array:
  _lock();

  // add the top sort at the highest index in this sort-code array
  // and set its bit code to be all ones for the relevant bit code
  // width:
  add(Sort.top());
  Sort.top().setIndex(size()-1).bitcode().set(0,size());

  // initialize the context of the bottom and top sorts:
  Sort.bottom().setContext(_context);
  Sort.top().setContext(_context);

  // initialize the decoded form of the bottom and top sorts:
  Sort.bottom().setDecoded(Decoded.bottom());
  Sort.top().setDecoded(Decoded.top());

  // register top and bottom in the named sort array and the code cache:
  _context.namedSorts().put(Sort.top().name(),Sort.top());
  _context.codeCache().put(Sort.top().bitcode(),Decoded.top());
  // System.err.println(">>> Committing sort \""+Sort.top()+"\" with decoded structure:");
  // System.err.println(Sort.top().decoded());
  // System.err.println("-------------------------------------------------------------");

  _context.namedSorts().put(Sort.bottom().name(),Sort.bottom());
  _context.codeCache().put(Sort.bottom().bitcode(),Decoded.bottom());
  // System.err.println(">>> Committing sort \""+Sort.bottom()+"\" with decoded structure:");
  // System.err.println(Sort.bottom().decoded());
  // System.err.println("-------------------------------------------------------------");

  // register the built-in sorts in the named sort array and the code
  // cache:
  for (int i=0; i<builtins.size(); i++)
    {
      Sort sort = getSort(i);
      _context.namedSorts().put(sort.name(),sort);
      Decoded decodedBuiltin = sort.decoded();
      _context.codeCache().put(sort.bitcode(),
                               decodedBuiltin != null
                               ? decodedBuiltin
                               : sort.setDecoded(new Decoded(sort)).decoded());
      // System.err.println(">>> Committing sort \""+sort+"\" with decoded structure:");
      // System.err.println(sort.decoded());
      // System.err.println("-------------------------------------------------------------");
    }
}

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

This verifies that there are no remaining sorts with unlocked codes. If there are any, this means that the taxonomy

contains cycles. In that case, this taxonomy is replaced with the unlocked sorts and re-encoded using the complete $O(n^3)$ transitive closure procedure, re-ordered, and swept for cycles. These cycles are reported by throwing `CyclicSortOrderingException` on the identified cycles.

```
void _checkCodes (int first, int last) throws CyclicSortOrderingException
{
  ArrayList badSorts = new ArrayList();

  for (int i= first; i<=last; i++)
    {
      Sort sort = getSort(i);
      if (!sort.isLocked())
        badSorts.add(sort);
    }

  if (badSorts.isEmpty())
    return;

  // There are sorts that are still unlocked repertoried in badSorts.

  // DisplayManager dm = _context.displayManager();
  // dm.println("*** "+(badSorts.size()-1)+
  //                  " sorts have not been properly encoded (cyclic taxonomy): "+
  //                  badSorts);

  reset();  // clear the taxonomy
  // replace the taxonomy with the "bad" sorts:
  int size = badSorts.size()-1;
  for (int index = 0; index<=size; index++)
    {
      Sort sort = (Sort)badSorts.get(index);
      sort.setIndex(index); // reset the index to the new one in taxonomy
      sort.bitcode().clear();    // erase all the bits in the sort's code
      sort.bitcode().set(index); // set the bit in this sort's bitcode corresponding to this sort's index
      add(sort);            // add the sort to the "bad sort" taxonomy
    }
  Sort.setCodeSize(size()); // reset the size of all codes to the size of this new taxonomy

  // Reinitialize the codes' bits from the "is-a" declarations; this
  // is done for a bad sort using only its unlocked parents. This
  // must be done after the loop refilling the taxonomy since we
  // need all the indices for the bad sorts to be all reset first
  // (which is done in the previous loop):
  for (int index = 0; index<=size; index++)
    {
      Sort sort = getSort(index);
      for (Iterator it=sort.parents().iterator(); it.hasNext();)
        {
          Sort parent = (Sort)it.next();
          if (!parent.isLocked())
            {
              // System.out.println(">>> "+sort+" <| "+parent);
              parent.bitcode().set(sort.index());
            }
        }
    }

  // perform "complete" Warshall transitive closure:
  _transitiveClosure(0,size);
  // reorder the "bad sort" taxonomy:
  _reorder(0,size);
  // perform cycle identification and throw a cyclic-ordering exception
  throw new CyclicSortOrderingException(_identifyCycles(0,size));
}
```

This is called when we know that the taxonomy contains cycles (and is therefore inconsistent). It will identify the

cycles in the reordered subset of inconsistent sorts closed with the complete transitive closure, all this having been done in the method above (_checkCodes).

```
private ArrayList _identifyCycles (int first, int last)
{
  _cycles.clear();
  ArrayList cycle = null;

  Sort sort = null;
  Sort prev = null;

  for (int i=first; i<=last; i++)
    {
      sort = getSort(i);

      if (prev != null && sort.bitcode().equals(prev.bitcode()))
        { // this is a cycle
          if (cycle == null)
            { // this cycle is new; create a record for it
              cycle = new ArrayList();
              // and initialize it with the previous sort
              cycle.add(prev.name());
            }
          // record the current sort as part of the cycle
          cycle.add(sort.name());
        }
      else
        { // this sort is not part of a cycle; make it the previous sort
          prev = sort;
          if (cycle != null)
            { // a complete cycle was detected: record it
              _cycles.add(cycle);
              // and reinitialize it for potential new cycles
              cycle = null;
            }
        }
    }

  // Sort.setCodeSize(size()+1);    // only for debugging purpose
  // showSortCodes(first,last);     // only for debugging purpose

  return _cycles;
}

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

This contains the numbers of sorts for each layer in this taxonomy. Therefore, its length is the height of the taxonomy.

```
IntArrayList layers = new IntArrayList();
```

This performs a transitive closure by assigning codes starting from bottom and proceeding upwards layer after layer until we reach top.

```
private void _bottomUpClosure ()
{
  HashSet layer = (HashSet)Sort.bottom().parents().clone();
  int height = 0;
```

```
    do
      {
        // System.out.println(">>> Encoding layer: "+layer);

        layers.add(layer.size());     // record the size of this layer
        if (Context.isTracing())
          System.out.println("*** Layer "+layers.size()+" has "+layer.size()+" sorts");

        _encodeLayer(layer);          // encode the layer
        layer = _nextLayer(layer);    // computes the next layer
      }
    while (!layer.isEmpty());
  }
```

This encodes all the sorts in the specified layer as the 'or' of its children, then locks it as encoded.

```
  private void _encodeLayer (HashSet layer)
  {
    for (Iterator it=layer.iterator(); it.hasNext();)
      {
        Sort sort = (Sort)it.next();
        for (Iterator i=sort.children().iterator(); i.hasNext();)
          sort.bitcode().or(((Sort)i.next()).bitcode());
        sort.lock();
      }
  }
```

The next layer is computed from the specified one as the union of all the parents of its elements from which are removed those elements that do not have all their children already encoded (*i.e.*, which still have at least one unlocked child).

```
  private HashSet _nextLayer (HashSet layer)
  {
    HashSet nextLayer = new HashSet();

    for (Iterator it=layer.iterator(); it.hasNext();)
      {
        HashSet parents = ((Sort)it.next()).parents();

        for (Iterator i=parents.iterator(); i.hasNext();)
          {
            Sort parent = (Sort)i.next();
            boolean skipParent = false;

            for (Iterator j=parent.children().iterator(); j.hasNext();)
              {
                Sort child = (Sort)j.next();
                if (!child.isLocked())
                  {
                    skipParent = true;
                    break;
                  }
              }

            if (!skipParent)
              nextLayer.add(parent);
          }
      }

    return nextLayer;
  }
```

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

This performs a Warshall transitive closure on the codes of the sorts stored in this taxonomy using the $O(n^3)$ Warshall's algorithm. This assumes that all the elements's codes have been set to their initial values as explained in the specification.

```
private void _transitiveClosure ()
{
  _transitiveClosure(0,size()-1);
}
```

This performs in-place transitive closure on the codes of the sorts stored in this taxonomy of all the elements between index first and index last (both inclusive).

```
private void _transitiveClosure (int first, int last)
{
  for (int k=first; k<=last; k++)
    for (int i=first; i<=last; i++)
      for (int j=first; j<=last; j++)
        {
          Sort sort_i = getSort(i);
          if (!sort_i.bitcode().get(j))
            sort_i.bitcode().set(j,sort_i.bitcode().get(k) && (getSort(k)).bitcode().get(j));
        }
}

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

This reorders in-place this taxonomy using the "precedes" comparison on Sort objects with (non-recursive) QuickSort. This assumes that all the elements' codes have their final values after transitive closure.

```
private void _reorder ()
{
  Misc.sort(this);
}
```

This reorders in-place the elements of this taxonomy between index first and index last (both inclusive) using the "precedes" comparison on Sort objects with (non-recursive) QuickSort. This assumes that all the elements' codes have their final values after transitive closure.

```
private void _reorder (int first, int last)
{
  Misc.sort(this,first,last);
}

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

Once all the sorts are defined in the taxonomy, we need to register each sort's name in the named-sort table and its code in the code cache.

```
private void _commitAllSorts (int first, int last)
{
  for (int i=first; i<=last; i++)
    _commitSort(i);
}
```

This returns the sort at the specified index in this taxonomy after registering its name and code in the named-sort table and its code in the code cache.

```
private Sort _commitSort (int index)
{
  Sort sort = getSort(index);

  _context.namedSorts().put(sort.name(),sort);
  _context.codeCache().put(sort.bitcode(),
                           sort.setDecoded(new Decoded(sort)).decoded());

  // System.err.println(">>> Committing sort \""+sort+"\" with decoded structure:");
  // System.err.println(sort.decoded());
  // System.err.println("-------------------------------------------------------------");

  return sort; // locking already done if using _bottomUpClosure()
  // return sort.lock(); // locking needed only if NOT using _bottomUpClosure()
}

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

This is used to record and report potential cycles that may be detected in the current taxonomy.

```
private ArrayList _cycles = new ArrayList();

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

Resets this entire taxonomy.

```
public void reset ()
{
  reset(0);
}
```

Resets this taxonomy for sorts of index higher or equal to `minIndex`.

Parameters: `minIndex` - the start index

```
public void reset (int minIndex)
{
  // clear the taxonomy above minIndex:
  clear(minIndex);
  // The following two hashset clearings are not fully correct in case
  // there are built-in sorts (all of index less than the size of the array
  // builtins) that need to be kept as children of top and parents of
  // bottom. NOTE TO MYSELF: not to forget on the to-do list Apr. 22, 2015
  Sort.top().children().clear();
  Sort.bottom().parents().clear();
}

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

This computes the set of maximum lower bounds of a given bit code as a bit code when the given code was obtained without negation. In that case, every "`1`" in the given bit code is at a position corresponding to the index of a lower bound of its sort. So, it is sufficient to loop through the `1`'s of the given bit code and set the corresponding position in the bitset "`glbs`" to "`1`" if it is maximal. **This is now rendered obsolete by the `Decode(BitCode)` method below.**

```
// public BitCode naiveGreatestLowerBounds (BitCode code) throws LockedCodeArrayException
// {
//   if (!_isLocked)
//  throw new LockedCodeArrayException("Attempt to perform an operation requiring prior sort encoding");

//   BitCode glbs = new BitCode();

//   Sort s = null;
//   Sort t = null;

//   for (IntIterator i = code.iterator(); i.hasNext();)
//  {
//    s = getSort(i.next());

//    // Skip top because it is not a lower bound:
//    if (s.isTop())
//      continue;

//    // We know by definition that s is a lower bound; but we need
//    // only the maximal ones. So, unless s has no supersort
//    // already in glbs, we must remove any subsort of s from glbs
//    // before adding s.

//    Boolean isMax = true;

//    for (IntIterator it = glbs.iterator(); it.hasNext();)
//      {
//        t = getSort(it.next());

//        if (s.bitcode().isStrictlyContainedIn(t.bitcode()))
//          { // t is a supersort of s; so s is not maximal: no need
//            // to proceed further:
//            isMax = false;
//            break;
//          }

//        if (t.bitcode().isStrictlyContainedIn(s.bitcode()))
//          // t is a subsort of s - remove it:
//          glbs.set(t.index(),false);
//      }

//    if (isMax)
//      glbs.set(s.index());
//  }

//   return glbs.lock();
// }
```

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

Decoding negated codes

The issue of decoding is critical: it must be as efficient as possible since it is used interactively. While such is possible for a code that has been computed without involving negation (see the `naiveGreatestLowerBounds` method above), it is less evident when negation is involved. The reason is that the semantics of a sort's code interprets the presence of a `1` in position *i* to mean that the sort of index *i* is a subsort of the one with this code, while a `0` means that it is *not a subsort*; that is, it must be either a supersort or incomparable. Hence, *taking the negation of a binary code is not symmetric*. Indeed, negating a code will yield a `1` where there was a `0`. Therefore, a `1` in position *i* in the code of a negated sort `!s` means that the corresponding sort *i* is either a supersort of, or incomparable with, `s`. This invalidates decoding such as the one performed by the `naiveGreatestLowerBounds` method above which keeps the maximal subsorts corresponding to position containing a `1` in the code and thus can only be used for codes obtained just using conjunction and disjunction, but no negation.

In the case where the code to be decoded was obtained using at least one negation, one alternative is to sweep the whole taxonomy and keep only maximal subsorts of each sorts. However, this is not viable as it can reach quadratic complexity in the size of the taxonomy.

Another possibility is to get back to the technique used in the `naiveGreatestLowerBounds` method, which involves only a loop through the `1` positions of a code. For this to be correct, it must be ensured that the codes respect the semantics whereby a `1` strictly indicates only a subsort.

In order to do that, let us consider the case of negating a sort expression `s`. If we compute the code of `!s` by switching all `1`s to `0`s and vice versa in the code of `s`, this will not work (as explained above). However, if we change a `1` in position *i* to a `0` in the code of `!s` whenever the sort of index *i* (*i.e.*, `TAXONOMY[i]`) is a supersort of a sort corresponding to a `0` in the code of `!s`, then we will be left with a code having a `1` only in positions of actual subsorts of `!s`. With such a code, the more efficient `naiveGreatestLowerBounds` method can then compute the correct set of maximal lower bounds of `!s`.

Note that when no negation has been used, the above method coincides with the `naiveGreatestLowerBounds` method. Indeed, in that case, there must be a `1` in all positions corresponding to subsorts (by construction, since such a code denotes the set of its subsorts). Hence, whenever there is a `0` in such a code, there is necessarily also a `0` in all the positions of its supersorts.

The method `greatestLowerBounds(BitCode code)` below implements the above scheme. **This is now rendered obsolete by the `Decode(BitCode)` method below.**

```
// public BitCode greatestLowerBounds (BitCode code) throws LockedCodeArrayException
// {
//   if (!_isLocked)
//  throw new LockedCodeArrayException("Attempt to perform an operation requiring prior sort encoding");

//   // we start by gathering the "undesirable" sorts; i.e., the
//   // ancestors of all the sorts corresponding to a 0 in the
//   // specified code bitcode - we iterate through the 0's in the
//   // code (or equivalently through the 1's in the negated code):

//   HashSet undesirableSet = new HashSet(code.size());

//   // System.out.println("\t>>> code = "+code);

//   System.out.print("\t>>> computing undesirables... ");

//   for (IntIterator it = BitCode.not(code).iterator(); it.hasNext();)
//  {
//    HashSet ancestors = getSort(it.next()).ancestors();
//    if (ancestors != null)
//      undesirableSet.addAll(ancestors);
//    // NB: note that we collect only strict ancestors - this will
//    // have for effect not to include the sort itself in the end
```

```
//    // if it is minimal.
//  }

//   System.out.println(undesirableSet.size()+" found.");

//   // System.out.println("\n>>> undesirables = "+undesirableSet);

//   // Define an initial set of lower bounds as a copy of code:
//   BitCode glbs = code.copy();

//   System.out.println("\t>>> removing the undesirables... ");

//   // Remove the undesirables from glbs:
//   for (Iterator it = undesirableSet.iterator(); it.hasNext();)
//  {
//    Sort s = (Sort)it.next();
//    glbs.clear(s.index());
//  }

//   System.out.println("\t>>> computing maximal lower bounds...");

//   // Next, we iterate through the 1's left in glbs and remove all
//   // their descendants, leaving only its maximal lower bounds:

//   for (IntIterator it = glbs.iterator(); it.hasNext();)
//  {
//    int subsortIndex = it.next();
//    glbs.andNot(getSort(subsortIndex).bitcode());
//    glbs.set(subsortIndex);
//  }

//   return glbs.lock();
// }
```

This returns a bit code representing the set of sorts whose codes are the minimal upper bounds of the specified bit code. Note that the returned bit code may be empty (if the code is the top sort's code). It iterates through the `0`s of the code in order to identify its supersorts, keeping only the minimals. **This is now rendered obsolete by the `Decode(BitCode)` method below.**

```
// public BitCode leastUpperBounds (BitCode code) throws LockedCodeArrayException, LockedBitCodeException
// {
//   if (!_isLocked)
//  throw new LockedCodeArrayException("Attempt to perform an operation requiring prior sort encoding");

//   BitCode lubs = new BitCode();

//   // iterate through the 0-bits of code:
//   for (IntIterator it = BitCode.not(code).iterator(); it.hasNext();)
//  {
//    int index = it.next();

//    Sort s = getSort(index);

//    if (!code.isStrictlyContainedIn(s.bitcode()))
//      // s is not a supersort - skip it:
//      continue;

//    // s is an uppper bound; however, lubs may contain supersorts
//    // of s; if so, we must remove them before adding this lower
//    // upper bound:

//    boolean isMinimal = true;

//    // proceed with the minimality check:
//    for (IntIterator i = lubs.iterator(); i.hasNext();)
//      {
//        Sort t = getSort(i.next());
```

```
//        if (t.bitcode().isStrictlyContainedIn(s.bitcode()))
//          { // s is a not a minimal supersort - break out of
//            // the minimality check loop:
//            isMinimal = false;
//            break;
//          }

//        if (s.bitcode().isStrictlyContainedIn(t.bitcode()))
//          {
//            System.out.println("\tremoving "+t);
//            lubs.remove(t.index());
//          }
//      }

//    if (isMinimal)
//      {
//        System.out.println("\tadding "+s);
//        lubs.add(index);
//      }
//  }

//   return lubs.lock();
// }

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

Notice that the two methods `greatestLowerBounds(BitCode)` and `leastUpperBounds(BitCode)` given above both loop through the `0`s of their `BitCode` argument. Since they are both systematically used for decoding a given code, it is a waste to go through such a loop twice, especially for a code containing a large number of `0`s. So, rather than using the `Decoded` constructor calling each method separately, it makes more sense to merge the two methods into a single one, building both glbs and lubs simultaneously.

The method `decode(BitCode)` returns a `Decoded` object with both lub and glb sets at once.

Parameters: `code` - a `BitCode`

Returns: a `Decoded` object for the specified bit code

```
public Decoded decode (BitCode code)
{
  if (!_isLocked)
    throw new LockedCodeArrayException("Attempt to perform an operation requiring prior sort encoding");

  // define an initial set of upper bounds as an empty bit code:
  BitCode lubs = new BitCode();
  // Define an initial set of lower bounds as a copy of code:
  BitCode glbs = code.copy();

  // in order to compute the glbs, we'll need to gather the "undesirable"
  // sorts - i.e., the ancestors of all the sorts corresponding to a 0 in
  // the specified code bitcode; we define an initial hash set for them:
  HashSet undesirableSet = new HashSet(code.size());

  // next, we iterate through the 0-bits of code (or equivalently through
  // the 1's in the negated code):
  for (IntIterator it = BitCode.not(code).iterator(); it.hasNext();)
    {
      int index = it.next();
```

```
        // we first take care of computing the undesirables for the
        // glbs in the case when negation was used (NB: note that we
        // collect only strict ancestors - this will have for effect
        // not to include the sort itself in the end if it is
        // minimal):
        if (Context.negativeQuery) // no need to do the following if no negation was used
          {
            Sort sort = getSort(index);
            // For debugging purposes:
            // System.err.println(">>> Sort = "+sort);

            HashSet ancestors = getSort(index).ancestors(this);

            if (ancestors != null)
              undesirableSet.addAll(ancestors);

            // For debugging purposes:
            // System.out.println("Computing undesirables: "+undesirableSet);
          }

        // we then take care of computing the lubs:
        Sort s = getSort(index);

        if (!code.isStrictlyContainedIn(s.bitcode()))
          // s is not a supersort - skip it:
          continue;

        // s is an uppper bound; however, lubs may contain supersorts
        // of s; if so, we must remove them before adding this lower
        // upper bound:

        boolean isMinimal = true;

        // proceed with the minimality check:
        for (IntIterator i = lubs.iterator(); i.hasNext();)
          {
            Sort t = getSort(i.next());

            if (t.bitcode().isStrictlyContainedIn(s.bitcode()))
              { // s is a not a minimal supersort - break out of
                // the minimality check loop:
                isMinimal = false;
                break;
              }

            if (s.bitcode().isStrictlyContainedIn(t.bitcode()))
              lubs.remove(t.index());
          }

        if (isMinimal)
          lubs.add(index);
      }

    // For debugging purposes:
    // System.out.println("Current glbs code is: "+glbs);

    // only in case negation was used do we need to compute the glbs using
    // the undesirables we just computed, first removing the undesirables
    // from glbs:
    if (Context.negativeQuery)
      {
        for (Iterator it = undesirableSet.iterator(); it.hasNext();)
          {
            Sort s = (Sort)it.next();
            glbs.clear(s.index());
            // For debugging purposes:
            // System.out.println("Removing undesirable sort: "+s+" (switching off bit at index: "+s.index()+")");
            // System.out.println("Remaining glbs code is: "+glbs);
          }

        // reset the negation detection:
```

```
      Context.negativeQuery = false;
    }

  // next, we iterate through the 1's left in glbs and remove all their
  // descendants, leaving only the greatest lower bounds:

  for (IntIterator it = glbs.iterator(); it.hasNext();)
    {
      int subsortIndex = it.next();
      glbs.andNot(getSort(subsortIndex).bitcode());
      glbs.set(subsortIndex);
    }

  // finally, we return the decoded object:
  return new Decoded(code,lubs.toHashSet(this),glbs.toHashSet(this));
}

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

Returns the height of the specified sort in its taxonomy. (See the specification.)

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

  int height = 0;

  for (IntIterator it = sort.bitcode().iterator(); it.hasNext();)
    {
      int index = it.next();

      if (index >= size())  // is this necessary???
        break;

      if (index == sort.index())
        continue;

      height = Math.max(height,computeHeight(getSort(index)));
    }

  return 1 + height;
}

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

This displays all the defined sorts in this taxonomy using the specified display manager.

```
public void showSortCodes ()
{
  showSortCodes(0,size()-2);
}
```

This displays the elements of this taxonomy between index first and index last (both inclusive) using the specified display manager. It always displays the top sort at the top (with no index) and the bottom sort at the bottom (with no index).

```
  public void showSortCodes (int first, int last)
  {
    int width = Misc.numWidth(last);
    DisplayManager dm = _context.displayManager();

    dm.println(Misc.repeat(width,' ')+" "+
               Sort.top().bitcode()+" "+
               Sort.top().name());

    for (int index=last; index>=first; index--)
      {
        Sort sort = getSort(index);
        dm.println(Misc.numberString(index,width)+" "+
                   sort.bitcode()+" "+
                   sort.name());
      }

    dm.println(Misc.repeat(width,' ')+" "+
               Sort.bottom().bitcode()+" "+
               Sort.bottom().name());
  }


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

This saves the current encoded taxonomy into a file on disk. The `String` argument is the name of the file. The format of this file is made out of two parts. The first part is such that the first line is the number of elements in the taxonomy (*i.e.*, `size()`), and each following line contains information for the sort `TAXONOMY[i]` for `i` ranging from `0` to `size()-1`. Each sort on a line is made of two parts:

`name code`

The `name` is the symbolic name of the sort. The `code` is the sort's `BitCode` code written as a space-separated sequence of pairs of integers, each number corresponding to the next true index followed by the next false index, going from the lowest to the highest. For example, the bitcode:

`00001111100111100000110000`

corresponds to the string:

`"4 6 12 16 18 23"`

The second part of the saved file consists of information to reconstruct the "is-a" ordering by saving the parents of each sort, from which "is-a" declarations can be processed at load time. This part consists of as many lines as there are non-maximal sorts, each line containing the index of a sort followed by the indices of the parents of sorts in the order of their indices in the taxonomy. **NB:** maximal sorts (*i.e.*, having top as a (necessarily single parent) are skipped. This is because there is no need to declare a subsort of top.

```
  public void save (String file) throws FileNotFoundException
  {
    PrintStream out = new PrintStream(file);

    int size = size()-1;      // no need to save the top sort
    out.println(size);

    Sort sort = null;

    // Saving the first part (each sort's name and code):
    for (int index=0; index<size; index++)
      {
        sort = getSort(index);

        out.print(sort.name());              // save the sort's name
        out.print(" ");
        out.println(sort.bitcode().toSaveFormatString());     // save the sort's code
```

```
      }

    // Saving the second part (each non-maximal sort's parents' indices):
    for (int index=0; index<size; index++)
      {
        sort = getSort(index);

        // skip this sort if it is maximal:
        if (sort.parents().contains(Sort.top()))
          continue;

        // save the sort's incex:
        out.print(index+" ");

        // System.out.print("\t>>> saving parents of "+sort+": ");

        // save the sort's parents' indices:
        for (Iterator it = sort.parents().iterator(); it.hasNext();)
          {
            Sort parent = (Sort)it.next();
            out.print(parent.index()+" ");
            // System.out.print(parent+" ("+parent.index()+") ");
          }

        out.println();
        // System.out.println();
      }

    out.close();
  }

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

This loads an encoded taxonomy that was saved in the specified file in the format explained in the `save` method, and initializes all the loaded sorts into the current context.

```
  public void load (String file) throws FileNotFoundException, IOException, BadTokenException
  {
    TaxonomyLoader input = new TaxonomyLoader(file);

    input.nextToken();        // read the taxonomy size (not counting top)

    if (input.tokenType() != input.NUMBER)
      input.error();

    // System.out.println(input.intValue());

    int size = input.intValue();

    // size+1 to account for the top sort to be added
    ensureCapacity(size+1);
    Sort.setCodeSize(size+1);

    input.nextToken();        // read the end of line

    Sort sort = null;
    BitCode code = null;
    int startBit = 0;

    for (int index=0; index<size; index++)
      {
        input.nextToken();    // read the symbol

        // System.out.print(input.symbolValue()+" ");

        if (input.tokenType() != input.WORD)
          input.error();
```

```
        // Create the sort and its code, and store the sort in the taxonomy at this index:
        code = new BitCode();
        sort = new Sort(index,input.symbolValue(),code).setContext(_context);
        set(index,sort);

        // read in the code and set the bits of the sort's code accordingly:

        input.nextToken();    // read the 1st "startBit" position

        while (input.tokenType() != input.EOL)
          {
            if (input.tokenType() != input.NUMBER)
              input.error();

            // System.out.print(input.intValue()+" ");
            startBit = input.intValue();

            input.nextToken();        // read the "endBit" position

            if (input.tokenType() != input.NUMBER)
              input.error();

            // System.out.print(input.intValue()+" ");
            code.set(startBit,input.intValue());

            input.nextToken();        // read either the next "startBit" position or the end of line
          }

        // install the sort as a named sort:
        _commitSort(index);
      }

    // Read each non-maximal sort's parents' indices and process the "is-a" information:
    for (;;)
      {
        input.nextToken();

        if (input.eof())
          break;

        if (input.tokenType() != input.NUMBER)
          input.error();

        int index = input.intValue();

        sort = getSort(index);

        // System.out.println(">>> Reading parents of "+sort+" (index = "+index+")");
        for (;;)
        // read and declare the parents:
          {
            // read the next parent index or end-of-line:
            input.nextToken();

            if (input.tokenType() == input.EOL)
              break;

            if (input.tokenType() != input.NUMBER)
              input.error();

            // process the is-a declaration:
            sort.addIsaDeclaration(getSort(input.intValue()));
            // System.out.println("\t>>> Declaring: "+sort+" <| "+getSort(input.intValue()));
          }

        sort.lock();
      }

    input.close();
    _initialize();
  }
```

```
    /* ************************************************************************ */
}
```

This file was generated on Thu Jul 11 06:49:14 PDT 2019 from file `Taxonomy.java`
by the `hlt.language.tools.Hilite` Java tool written by Hassan Aït-Kaci
