```
/* License (BSD Style License):
package saere.predicate;

import static saere.StringAtom.EMPTY_LIST;
import static saere.StringAtom.LIST;
import static saere.term.Terms.complexTerm;
import static saere.term.Terms.variable;
import saere.Goal;
import saere.GoalStack;
import saere.PredicateIdentifier;
import saere.PredicateRegistry;
import saere.State;
import saere.StringAtom;
import saere.Term;
import saere.ThreeArgsPredicateFactory;

/**
 * Implementation of ISO Prolog's append/3 predicate.
 *
 * <pre>
 * <code>
 * append([],Ys,Ys).
 * append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
 * </code>
 * </pre>
 *
 * @author Michael Eichberg (mail@michael-eichberg.de)
 */
public final class Append3 implements Goal {

    public final static PredicateIdentifier IDENTIFIER = new
PredicateIdentifier(
            StringAtom.get("append"), 3);

    public final static ThreeArgsPredicateFactory FACTORY = new
ThreeArgsPredicateFactory() {

        @Override
        public Goal createInstance(Term t1, Term t2, Term t3) {
            return new Append3(t1, t2, t3);
        }

    };

    public static void registerWithPredicateRegistry(PredicateRegistry
```

```
registry) {
        registry.register(IDENTIFIER, FACTORY);
    }

    // variables to control/manage the execution this predicate
    private int clauseToExecute = 1;
    private GoalStack goalStack = GoalStack.emptyStack();
    private int goalToExecute = 1;
    // variables related to the predicate's state
    private Term arg0;
    private Term arg1;
    private Term arg2;
    final private State arg0State; // REQUIRED BY TAIL-CALL
OPTIMIZATION ...
    final private State arg1State;
    final private State arg2State;
    // variables to store clause local information
    private Term clv0;
    private Term clv1;
    private Term clv2;

    public Append3(final Term arg0, final Term arg1, final Term arg2)
{
        this.arg0 = arg0;
        this.arg1 = arg1;
        this.arg2 = arg2;

        this.arg0State = arg0.manifestState(); // REQUIRED BY TAIL-
CALL OPTIMIZATION ...
        this.arg1State = arg1.manifestState();
        this.arg2State = arg2.manifestState();
    }

    public void abort() {
        this.goalStack = goalStack.abortPendingGoals();
    }

    public boolean choiceCommitted() {
        return false;
    }

    public boolean next() {
        do { // REQUIRED BY TAIL-CALL OPTIMIZATION ...
            switch (this.clauseToExecute) {
            case 1: {
```

```
                if (this.clause1()) {
                    return true;
                } else {
                    // this clause contains no "cut"
                    // prepare the execution of the next clause
                    this.goalToExecute = 1;
                    this.clauseToExecute = 2;
                }
            }
            case 2: {
                // REQUIRED BY TAIL-CALL OPTIMIZATION ...
                if(this.clause2()) {
                    continue;
                } else {
                    arg0State.reincarnate();
                    arg1State.reincarnate();
                    arg2State.reincarnate();
                    return false;
                }
            }
            default:
                // should never be reached
                throw new Error("internal compiler error");
            }
        } while (true);
    }

    private boolean clause1() {
        eval_goals: do {
            switch (this.goalToExecute) {
            case 1: {
                this.goalStack = goalStack.put(new Unify2(arg0,
EMPTY_LIST));
            }
            case 2: {
                boolean succeeded = this.goalStack.peek().next();
                if (!succeeded) {
                    this.goalStack = goalStack.drop();
                    return false;
                }
                // fall through ... 3
            }
            case 3: {
                this.goalStack = goalStack.put(new Unify2(arg2,
arg1));
```

```
            }
            case 4: {
                boolean succeeded = this.goalStack.peek().next();
                if (!succeeded) {
                    this.goalStack = goalStack.drop();
                    this.goalToExecute = 2;
                    continue eval_goals;
                }
                this.goalToExecute = 4;
                return true;
            }
            default:
                // should never be reached
                throw new Error("internal compiler error");
            }
        } while (true);
    }

    private boolean clause2() {
        eval_goals: do {
            switch (this.goalToExecute) {
            case 1: {
                this.clv0 = variable();
                this.clv1 = variable();
                this.goalStack = goalStack.put(new Unify2(arg0,
complexTerm(
                        LIST, clv0, clv1)));
            }
            case 2: {
                boolean succeeded = this.goalStack.peek().next();
                if (!succeeded) {
                    this.goalStack = goalStack.drop();
                    return false;
                }
                // fall through ... 3
            }
            case 3: {
                this.clv2 = variable();
                this.goalStack = goalStack.put(new Unify2(arg2,
complexTerm(
                        LIST, clv0, clv2)));
            }
            case 4: {
                boolean succeeded = this.goalStack.peek().next();
                if (!succeeded) {
```

```
                    this.goalStack = goalStack.drop();
                    this.goalToExecute = 2;
                    continue eval_goals;
                }
                // fall through ... 5
            }
            case 5: {
                // update input variables
                arg0 = clv1;
                //arg1 = arg1;
                arg2 = clv2;
                // prepare next round...
                this.clauseToExecute = 1;
                this.goalToExecute = 1;
                this.goalStack = GoalStack.emptyStack();
                return true;
//
//              this.goalStack = goalStack.put(new append3(clv1, arg1,
clv2));
//          }
//          case 6: {
//              boolean succeeded = this.goalStack.peek().next();
//              if (!succeeded) {
//                  this.goalStack = goalStack.drop();
//                  this.goalToExecute = 4;
//                  continue eval_goals;
//              }
//              this.goalToExecute = 6;
//              return true;
            }
            default:
                // should never be reached
                throw new Error("internal compiler error");
            }
        } while (true);
    }

    /*
     *
     * // variables to control/manage the execution this predicate
private
     * boolean baseCaseClause = true; private boolean call = true;
     *
     *
     * // variables related to the predicate's state final private
```

```
Term Xs;
     * final private Term Ys; final private Term Zs; private State
XsState;
     * private State YsState; private State ZsState;
     *
     * public Append3(final Term Xs, final Term Ys,final Term Zs)
{ this.Xs =
     * Xs.unwrapped(); this.Ys = Ys; this.Zs = Zs.unwrapped(); }
     *
     * public boolean next() { if (baseCaseClause) { if (this.clause1
()) {
     * return true; } else { this.baseCaseClause = false; } } return
     * this.clause2(); }
     *
     *
     * private boolean clause1() { if (call) { if(Xs.isVariable())
{ XsState =
     * Xs.manifestState(); Xs.asVariable().bind
(StringAtom.EMPTY_LIST); } else
     * if (Xs != StringAtom.EMPTY_LIST) { return false; }
     *
     * YsState = Ys.manifestState(); ZsState = Zs.manifestState();
     * if(Ys.unify(Zs)){ return true; } } if (XsState != null)
     * XsState.reincarnate(); if (YsState != null) YsState.reincarnate
(); if
     * (ZsState != null) ZsState.reincarnate(); return false; }
     *
     *
     * @Override public void abort() { throw new Error(); }
     *
     * @Override public boolean choiceCommitted() { return false; }
     */
}
```