```java
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import java.util.Hashtable;
import static jminusminus.CLConstants.*;

/**
 * The AST for an cast expression, which has both a cast (a type) and the
 * expression to be cast.
 */

class JCastOp extends JExpression {

    /** The cast. */
    private Type cast;

    /** The expression we're casting. */
    private JExpression expr;

    /** The conversions table. */
    private static Conversions conversions;

    /** The converter to use for this cast. */
    private Converter converter;

    /**
     * Construct an AST node for a cast operation from its line number, cast,
     * and expression.
     *
     * @param line
     *            the line in which the operation occurs in the source file.
     * @param cast
     *            the type we're casting our expression as.
     * @param expr
     *            the expression we're casting.
     */

    public JCastOp(int line, Type cast, JExpression expr) {
        super(line);
        this.cast = cast;
        this.expr = expr;
        conversions = new Conversions();
    }

    /**
     * Analyzing a cast expression means, resolving the type (to which we are
     * casting), checking the legality of the cast, and computing a (possibly
     * null) conversion for use in code generation.
     *
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JExpression analyze(Context context) {
        expr = (JExpression) expr.analyze(context);
        type = cast = cast.resolve(context);
        if (cast.equals(expr.type())) {
            converter = Converter.Identity;
        } else if (cast.isJavaAssignableFrom(expr.type())) {
            converter = Converter.WidenReference;
        } else if (expr.type().isJavaAssignableFrom(cast)) {
            converter = new NarrowReference(cast);
        } else if ((converter = conversions.get(expr.type(), cast)) != null) {
        } else {
```

```
67              JAST.compilationUnit.reportSemanticError(line, "Cannot cast a "
68                      + expr.type().toString() + " to a " + cast.toString());
69          }
70          return this;
71      }
72
73      /**
74       * Generating code for a cast expression involves generating code for the
75       * original expr, and then for any necessary conversion.
76       *
77       * @param output
78       *             the code emitter (basically an abstraction for producing the
79       *             .class file).
80       */
81
82      public void codegen(CLEmitter output) {
83          expr.codegen(output);
84          converter.codegen(output);
85      }
86
87      /**
88       * @inheritDoc
89       */
90
91      public void writeToStdOut(PrettyPrinter p) {
92          p.printf("<JCastOp line=\"%d\" type=\"%s\"/>\n", line(),
93                  ((cast == null) ? "" : cast.toString()));
94          p.indentRight();
95          if (expr != null) {
96              p.printf("<Expression>\n");
97              p.indentRight();
98              expr.writeToStdOut(p);
99              p.indentLeft();
100             p.println("</Expression>");
101         }
102         p.indentLeft();
103         p.println("</JCastOp>");
104     }
105
106 }
107
108 /**
109  * A 2-D table of conversions, from one type to another.
110  */
111
112 class Conversions {
113
114     /**
115      * Table of conversions; maps a source and target type pair to its
116      * converter.
117      */
118     private Hashtable<String, Converter> table;
119
120     /**
121      * Construct a table of conversions and populate it.
122      */
123
124     public Conversions() {
125         table = new Hashtable<String, Converter>();
126
127         // Populate the table
128
129         put(Type.CHAR, Type.INT, Converter.Identity);
130         put(Type.INT, Type.CHAR, new I2C());
131
132         // Boxing
133         put(Type.CHAR, Type.BOXED_CHAR, new Boxing(Type.CHAR, Type.BOXED_CHAR));
134         put(Type.INT, Type.BOXED_INT, new Boxing(Type.INT, Type.BOXED_INT));
135         put(Type.BOOLEAN, Type.BOXED_BOOLEAN, new Boxing(Type.BOOLEAN,
```

```
                Type.BOXED_BOOLEAN));

        // Un-boxing
        put(Type.BOXED_CHAR, Type.CHAR, new UnBoxing(Type.BOXED_CHAR,
                Type.CHAR, "charValue"));
        put(Type.BOXED_INT, Type.INT, new UnBoxing(Type.BOXED_INT, Type.INT,
                "intValue"));
        put(Type.BOXED_BOOLEAN, Type.BOOLEAN, new UnBoxing(Type.BOXED_BOOLEAN,
                Type.BOOLEAN, "booleanValue"));
    }

    /**
     * Define a conversion. This is used locally, for populating the table.
     *
     * @param source
     *            the original type.
     * @param target
     *            the target type.
     * @param c
     *            the converter necessary.
     */

    private void put(Type source, Type target, Converter c) {
        table.put(source.toDescriptor() + "2" + target.toDescriptor(), c);
    }

    /**
     * Retrieve a converter for converting from some original type to a target
     * type; the converter may be empty (requiring no code for run-time
     * execution).
     *
     * @param source
     *            the original type.
     * @param target
     *            the target type.
     * @return the converter.
     */

    public Converter get(Type source, Type target) {
        return table.get(source.toDescriptor() + "2" + target.toDescriptor());
    }

}

/**
 * A Converter encapusates any (possibly none) code necessary to perform a cast
 * operation.
 */

interface Converter {

    /** For identity conversion (no run-time code needed). */
    public static Converter Identity = new Identity();

    /** For widening conversion (no run-time code needed). */
    public static Converter WidenReference = Identity;

    /**
     * Emit code necessary to convert (cast) a source type to a target type.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output);

}
```

```
/**
 * The identity conversion requires no run-time code.
 */

class Identity implements Converter {

    /**
     * @inheritDoc
     */

    public void codegen(CLEmitter output) {
        // Nothing
    }

}

/**
 * A narrowing conversion on a reference type requires a run-time check on the
 * type.
 */

class NarrowReference implements Converter {

    /** The target type. */
    private Type target;

    /**
     * Construct a narrowing converter.
     *
     * @param target
     *            the target type.
     */

    public NarrowReference(Type target) {
        this.target = target;
    }

    /**
     * @inheritDoc
     */

    public void codegen(CLEmitter output) {
        output.addReferenceInstruction(CHECKCAST, target.jvmName());
    }

}

/**
 * Boxing requires invoking the appropriate conversion method from the (Java)
 * API.
 */

class Boxing implements Converter {

    /** The source type. */
    private Type source;

    /** The target type. */
    private Type target;

    /**
     * Construct a Boxing converter.
     *
     * @param source
     *            the source type.
     * @param target
     *            the target type.
     */

```

```
274     public Boxing(Type source, Type target) {
275         this.source = source;
276         this.target = target;
277     }
278 
279     /**
280      * @inheritDoc
281      */
282 
283     public void codegen(CLEmitter output) {
284         output.addMemberAccessInstruction(INVOKESTATIC, target.jvmName(),
285                 "valueOf", "(" + source.toDescriptor() + ")"
286                         + target.toDescriptor());
287     }
288 
289 }
290 
291 /**
292  * Unboxing requires invoking the appropriate conversion method from the (Java)
293  * API.
294  */
295 
296 class UnBoxing implements Converter {
297 
298     /** The source type. */
299     private Type source;
300 
301     /** The target type. */
302     private Type target;
303 
304     /** The (Java) method to invoke for the conversion. */
305     private String methodName;
306 
307     /**
308      * Construct an UnBoxing converter.
309      *
310      * @param source
311      *             the source type.
312      * @param target
313      *             the target type.
314      * @param methodName
315      *             the (Java) method to invoke for the conversion.
316      */
317 
318     public UnBoxing(Type source, Type target, String methodName) {
319         this.source = source;
320         this.target = target;
321         this.methodName = methodName;
322     }
323 
324     /**
325      * @inheritDoc
326      */
327 
328     public void codegen(CLEmitter output) {
329         output.addMemberAccessInstruction(INVOKEVIRTUAL, source.jvmName(),
330                 methodName, "()" + target.toDescriptor());
331     }
332 
333 }
334 
335 /**
336  * Converting from an int to a char requires an I2C instruction.
337  */
338 
339 class I2C implements Converter {
340 
341     /**
342      * @inheritDoc
```

```
343      */
344
345     public void codegen(CLEmitter output) {
346         output.addNoArgInstruction(I2C);
347     }
348
349 }
350
```