```
1  // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import java.lang.reflect.Array;
6  import java.lang.reflect.Modifier;
7  import java.util.Arrays;
8  import java.util.ArrayList;
9  import java.util.Hashtable;
10
11 /**
12  * For representing j-- types. All types are represented underneath (in the
13  * classRep field) by Java objects of type Class. These ojects represent types
14  * in Java, so this should ease our interfacing with existing Java classes.
15  *
16  * Class types (reference types that are represented by the identifiers
17  * introduced in class declarations) are represented using TypeName. So for now,
18  * every TypeName represents a class. In the future, TypeName could be extended
19  * to represent interfaces or enumerations.
20  *
21  * IdentifierTypes must be "resolved" at some point, so that all Types having
22  * the same name refer to the same Type object. resolve() does this.
23  */
24
25 class Type {
26
27     /** The Type's internal (Java) representation. */
28     private Class<?> classRep;
29
30     /** Maps type names to their Type representations. */
31     private static Hashtable<String, Type> types = new Hashtable<String, Type>();
32
33     /** The primitive type, int. */
34     public final static Type INT = typeFor(int.class);
35
36     /** The primitive type, char. */
37     public final static Type CHAR = typeFor(char.class);
38
39     /** The primitive type, boolean. */
40     public final static Type BOOLEAN = typeFor(boolean.class);
41
42     /** java.lang.Integer. */
43     public final static Type BOXED_INT = typeFor(java.lang.Integer.class);
44
45     /** java.lang.Character. */
46     public final static Type BOXED_CHAR = typeFor(java.lang.Character.class);
47
48     /** java.lang.Boolean. */
49     public final static Type BOXED_BOOLEAN = typeFor(java.lang.Boolean.class);
50
51     /** The type java.lang.String. */
52     public static Type STRING = typeFor(java.lang.String.class);
53
54     /** The type java.lang.Object. */
55     public static Type OBJECT = typeFor(java.lang.Object.class);
56
57     /** The void type. */
58     public final static Type VOID = typeFor(void.class);
59
60     /** The null void. */
61     public final static Type NULLTYPE = new Type(java.lang.Object.class);
62
63     /**
64      * A type marker indicating a constructor (having no return type).
65      */
66     public final static Type CONSTRUCTOR = new Type(null);
```

```
67
68      /** The "any" type (denotes wild expressions). */
69      public final static Type ANY = new Type(null);
70
71      /**
72       * Construct a Type representation for a type from its Java (Class)
73       * representation. Use typeFor() -- that maps types having like classReps to
74       * like Types.
75       *
76       * @param classRep
77       *            the Java representation.
78       */
79
80      private Type(Class<?> classRep) {
81          this.classRep = classRep;
82      }
83
84      /** This constructor is to keep the compiler happy. */
85
86      protected Type() {
87          super();
88      }
89
90      /**
91       * Construct a Type representation for a type from its (Java) Class
92       * representation. Make sure there is a unique Type for each unique type.
93       *
94       * @param classRep
95       *            the Java representation.
96       */
97
98      public static Type typeFor(Class<?> classRep) {
99          if (types.get(descriptorFor(classRep)) == null) {
100             types.put(descriptorFor(classRep), new Type(classRep));
101         }
102         return types.get(descriptorFor(classRep));
103     }
104
105     /**
106      * Return the class representation for a type, appropriate for dealing with
107      * the Java reflection API.
108      *
109      * @return the Class representation for this type.
110      */
111
112     public Class<?> classRep() {
113         return classRep;
114     }
115
116     /**
117      * This setter is used by JCompilationUnit.preAnalyze() to set the classRep
118      * to the specified partial class, computed during pre-analysis.
119      *
120      * @param classRep
121      *            the partial class.
122      */
123
124     public void setClassRep(Class<?> classRep) {
125         this.classRep = classRep;
126     }
127
128     /**
129      * Type equality is based on the equality of descriptors.
130      *
131      * @param that
132      *            the other Type.
133      * @return true iff the two types are equal.
134      */
135
```

```
136     public boolean equals(Type that) {
137         return this.toDescriptor().equals(that.toDescriptor());
138     }
139
140     /**
141      * Is this an Array type?
142      *
143      * @return true or false.
144      */
145
146     public boolean isArray() {
147         return classRep.isArray();
148     }
149
150     /**
151      * An array type's component type. Meaningful only for array types.
152      *
153      * @return the component type.
154      */
155
156     public Type componentType() {
157         return typeFor(classRep.getComponentType());
158     }
159
160     /**
161      * Return the Type's super type (or null if there is none). Meaningful only
162      * to class Types.
163      *
164      * @return the super type.
165      */
166
167     public Type superClass() {
168         return classRep == null || classRep.getSuperclass() == null ? null
169                 : typeFor(classRep.getSuperclass());
170     }
171
172     /**
173      * Is this a primitive type?
174      *
175      * @return true or false.
176      */
177
178     public boolean isPrimitive() {
179         return classRep.isPrimitive();
180     }
181
182     /**
183      * Is this an interface type?
184      *
185      * @return true or false.
186      */
187
188     public boolean isInterface() {
189         return classRep.isInterface();
190     }
191
192     /**
193      * Is this a reference type?
194      *
195      * @return true or false.
196      */
197
198     public boolean isReference() {
199         return !isPrimitive();
200     }
201
202     /**
203      * Is this type declared final?
204      *
```

```
205      * @return true or false.
206      */
207
208     public boolean isFinal() {
209         return Modifier.isFinal(classRep.getModifiers());
210     }
211
212     /**
213      * Is this type declared abstract?
214      *
215      * @return true or false.
216      */
217
218     public boolean isAbstract() {
219         return Modifier.isAbstract(classRep.getModifiers());
220     }
221
222     /**
223      * Is this a supertype of that?
224      *
225      * @param that
226      *             the candidate subtype.
227      * @return true iff this is a supertype of that.
228      */
229
230     public boolean isJavaAssignableFrom(Type that) {
231         return this.classRep.isAssignableFrom(that.classRep);
232     }
233
234     /**
235      * Return a list of this class' abstract methods? It does has abstract
236      * methods if (1) Any method declared in the class is abstract, or (2) Its
237      * superclass has an abstract method which is not overridden here.
238      *
239      * @return a list of abstract methods
240      */
241
242     public ArrayList<Method> abstractMethods() {
243         ArrayList<Method> inheritedAbstractMethods = superClass() == null ? new
ArrayList<Method>()
244                 : superClass().abstractMethods();
245         ArrayList<Method> abstractMethods = new ArrayList<Method>();
246         ArrayList<Method> declaredConcreteMethods = declaredConcreteMethods();
247         ArrayList<Method> declaredAbstractMethods = declaredAbstractMethods();
248         abstractMethods.addAll(declaredAbstractMethods);
249         for (Method method : inheritedAbstractMethods) {
250             if (!declaredConcreteMethods.contains(method)
251                     && !declaredAbstractMethods.contains(method)) {
252                 abstractMethods.add(method);
253             }
254         }
255         return abstractMethods;
256     }
257
258     /**
259      * Return a list of this class' declared abstract methods.
260      *
261      * @return a list of declared abstract methods.
262      */
263
264     private ArrayList<Method> declaredAbstractMethods() {
265         ArrayList<Method> declaredAbstractMethods = new ArrayList<Method>();
266         for (java.lang.reflect.Method method : classRep.getDeclaredMethods()) {
267             if (Modifier.isAbstract(method.getModifiers())) {
268                 declaredAbstractMethods.add(new Method(method));
269             }
270         }
271         return declaredAbstractMethods;
272     }
```

```
273
274     /**
275      * Return a list of this class' declared concrete methods.
276      *
277      * @return a list of declared concrete methods.
278      */
279
280     private ArrayList<Method> declaredConcreteMethods() {
281         ArrayList<Method> declaredConcreteMethods = new ArrayList<Method>();
282         for (java.lang.reflect.Method method : classRep.getDeclaredMethods()) {
283             if (!Modifier.isAbstract(method.getModifiers())) {
284                 declaredConcreteMethods.add(new Method(method));
285             }
286         }
287         return declaredConcreteMethods;
288     }
289
290     /**
291      * An assertion that this type matches one of the specified types. If there
292      * is no match, an error message is returned.
293      *
294      * @param line
295      *            the line near which the mismatch occurs.
296      * @param expectedTypes
297      *            expected types.
298      */
299
300     public void mustMatchOneOf(int line, Type... expectedTypes) {
301         if (this == Type.ANY)
302             return;
303         for (int i = 0; i < expectedTypes.length; i++) {
304             if (matchesExpected(expectedTypes[i])) {
305                 return;
306             }
307         }
308         JAST.compilationUnit.reportSemanticError(line,
309                 "Type %s doesn't match any of the expected types %s", this,
310                 Arrays.toString(expectedTypes));
311     }
312
313     /**
314      * An assertion that this type matches the specified type. If there is no
315      * match, an error message is written.
316      *
317      * @param line
318      *            the line near which the mismatch occurs.
319      * @param expectedType
320      *            type with which to match.
321      */
322
323     public void mustMatchExpected(int line, Type expectedType) {
324         if (!matchesExpected(expectedType)) {
325             JAST.compilationUnit.reportSemanticError(line,
326                     "Type %s doesn't match type %s", this, expectedType);
327         }
328     }
329
330     /**
331      * Does this type match the expected type? For now, "matches" means
332      * "equals".
333      *
334      * @param expected
335      *            the type that this might match.
336      * @return true or false.
337      */
338
339     public boolean matchesExpected(Type expected) {
340         return this == Type.ANY || expected == Type.ANY
341                 || (this == Type.NULLTYPE && expected.isReference())
```

```
                || this.equals(expected);
    }

    /**
     * Do argument types match? A helper used for finding candidate methods and
     * constructors.
     *
     * @param argTypes1
     *            arguments (classReps) of one method.
     * @param argTypes2
     *            arguments (classReps) of another method.
     * @return true iff all corresponding types of argTypes1 and argTypes2
     *         match.
     */

    public static boolean argTypesMatch(Class<?>[] argTypes1,
            Class<?>[] argTypes2) {
        if (argTypes1.length != argTypes2.length) {
            return false;
        }
        for (int i = 0; i < argTypes1.length; i++) {
            if (!Type.descriptorFor(argTypes1[i]).equals(
                    Type.descriptorFor(argTypes2[i]))) {
                return false;
            }
        }
        return true;
    }

    /**
     * Return the simple (unqualified) name for this type. Eg, String in place
     * of java.lang.String.
     *
     * @return the simple name.
     */

    public String simpleName() {
        return classRep.getSimpleName();
    }

    /**
     * A printable (j--) string representation of this type. Eg, int[],
     * java.lang.String.
     *
     * @return the string representation.
     */

    public String toString() {
        return toJava(this.classRep);
    }

    /**
     * The JVM descriptor for this type. Eg, Ljava/lang/String; for
     * java.lang.String, [[Z for boolean[][].
     *
     * @return the descriptor.
     */

    public String toDescriptor() {
        return descriptorFor(classRep);
    }

    /**
     * A helper translating a type's internal representation to its (JVM)
     * descriptor.
     *
     * @param cls
     *            internal representation whose descriptor is required.
     * @return the JVM descriptor.
```

```
411      */
412
413     private static String descriptorFor(Class<?> cls) {
414         return cls == null ? "V" : cls == void.class ? "V"
415                 : cls.isArray() ? "[" + descriptorFor(cls.getComponentType())
416                         : cls.isPrimitive() ? (cls == int.class ? "I"
417                                 : cls == char.class ? "C"
418                                         : cls == boolean.class ? "Z" : "?")
419                                 : "L" + cls.getName().replace('.', '/') + ";";
420     }
421
422     /**
423      * The JVM representation for this type's name. This is also called the
424      * internal form of the name. Eg, java/lang/String for java.lang.String.
425      *
426      * @return the type's name in internal form.
427      */
428
429     public String jvmName() {
430         return this.isArray() || this.isPrimitive() ? this.toDescriptor()
431                 : classRep.getName().replace('.', '/');
432     }
433
434     /**
435      * Return the Java (and so j--) denotation for the specified type. Eg,
436      * int[], java.lang.String.
437      *
438      * @param classRep
439      *            the internal representation of type whose Java denotation is
440      *            required
441      * @return the Java denotation.
442      */
443
444     private static String toJava(Class classRep) {
445         return classRep.isArray() ? toJava(classRep.getComponentType()) + "[]"
446                 : classRep.getName();
447     }
448
449     /**
450      * Return the type's package name. Eg, java.lang for java.lang.String.
451      *
452      * @return the package name.
453      */
454
455     public String packageName() {
456         String name = toString();
457         return name.lastIndexOf('.') == -1 ? "" : name.substring(0, name
458                 .lastIndexOf('.') - 1);
459     }
460
461     /**
462      * The String representation for a type being appended to a StringBuffer for
463      * + and += over strings.
464      *
465      * @return a string representation of the type.
466      */
467
468     public String argumentTypeForAppend() {
469         return this == Type.STRING || this.isPrimitive() ? toDescriptor()
470                 : "Ljava/lang/Object;";
471     }
472
473     /**
474      * Find an appropriate method in this type, given a message (method) name
475      * and it's argument types. This is pretty easy given our (current)
476      * restriction that the types of the actual arguments must exactly match the
477      * types of the formal parameters. Returns null if it cannot find one.
478      *
479      * @param name
```

```
480      *            the method name.
481      * @param argTypes
482      *            the argument types.
483      * @return Method with given name and argument types, or null.
484      */
485
486     public Method methodFor(String name, Type[] argTypes) {
487         Class[] classes = new Class[argTypes.length];
488         for (int i = 0; i < argTypes.length; i++) {
489             classes[i] = argTypes[i].classRep;
490         }
491         Class cls = classRep;
492
493         // Search this class and all superclasses
494         while (cls != null) {
495             java.lang.reflect.Method[] methods = cls.getDeclaredMethods();
496             for (java.lang.reflect.Method method : methods) {
497                 if (method.getName().equals(name)
498                         && Type.argTypesMatch(classes, method
499                                 .getParameterTypes())) {
500                     return new Method(method);
501                 }
502             }
503             cls = cls.getSuperclass();
504         }
505         return null;
506     }
507
508     /**
509      * Find an appropriate constructor in this type, given its argument types.
510      * This is pretty easy given our (current) restriction that the types of the
511      * actual arguments must exactly match the types of the formal parameters.
512      * Returns null if it cannot find one.
513      *
514      * @param argTypes
515      *            the argument types.
516      * @return Constructor with the specified argument types, or null.
517      */
518
519     public Constructor constructorFor(Type[] argTypes) {
520         Class[] classes = new Class[argTypes.length];
521         for (int i = 0; i < argTypes.length; i++) {
522             classes[i] = argTypes[i].classRep;
523         }
524
525         // Search only this class (we don't inherit constructors)
526         java.lang.reflect.Constructor[] constructors = classRep
527                 .getDeclaredConstructors();
528         for (java.lang.reflect.Constructor constructor : constructors) {
529             if (argTypesMatch(classes, constructor.getParameterTypes())) {
530                 return new Constructor(constructor);
531             }
532         }
533         return null;
534     }
535
536     /**
537      * Return the Field having this name.
538      *
539      * @param name
540      *            the name of the field we want.
541      * @return the Field or null if it's not there.
542      */
543
544     public Field fieldFor(String name) {
545         Class<?> cls = classRep;
546         while (cls != null) {
547             java.lang.reflect.Field[] fields = cls.getDeclaredFields();
548             for (java.lang.reflect.Field field : fields) {
```

```
                if (field.getName().equals(name)) {
                    return new Field(field);
                }
            }
            cls = cls.getSuperclass();
        }
        return null;
    }

    /**
     * Convert an array of argument types to a string representation of a
     * parenthesized list of the types, eg, (int, boolean, java.lang.String).
     *
     * @param argTypes
     *            the array of argument types.
     * @return the string representation.
     */

    public static String argTypesAsString(Type[] argTypes) {
        if (argTypes.length == 0) {
            return "()";
        } else {
            String str = "(" + argTypes[0].toString();
            for (int i = 1; i < argTypes.length; i++) {
                str += "," + argTypes[i];
            }
            str += ")";
            return str;
        }
    }

    /**
     * Check the accessibility of a member from this type (that is, this type is
     * the referencing type).
     *
     * @param line
     *            the line in which the access occurs.
     * @param member
     *            the member being accessed.
     * @return true if access is valid; false otherwise.
     */

    public boolean checkAccess(int line, Member member) {
        if (!checkAccess(line, classRep, member.declaringType().classRep)) {
            return false;
        }

        // Secondly, the member must be either public,
        if (member.isPublic()) {
            return true;
        }
        java.lang.Package p1 = classRep.getPackage();
        java.lang.Package p2 = member.declaringType().classRep.getPackage();
        if ((p1 == null ? "" : p1.getName()).equals((p2 == null ? "" : p2
                .getName()))) {
            return true;
        }
        if (member.isProtected()) {
            if (classRep.getPackage().getName().equals(
                    member.declaringType().classRep.getPackage().getName())
                    || typeFor(member.getClass().getDeclaringClass())
                            .isJavaAssignableFrom(this)) {
                return true;
            } else {
                JAST.compilationUnit.reportSemanticError(line,
                        "The protected member, " + member.name()
                                + ", is not accessible.");
                return false;
            }
```

```
618         }
619         if (member.isPrivate()) {
620             if (descriptorFor(classRep).equals(
621                     descriptorFor(member.member().getDeclaringClass()))) {
622                 return true;
623             } else {
624                 JAST.compilationUnit.reportSemanticError(line,
625                         "The private member, " + member.name()
626                                 + ", is not accessible.");
627                 return false;
628             }
629         }
630 
631         // Otherwise, the member has default access
632         if (packageName().equals(member.declaringType().packageName())) {
633             return true;
634         } else {
635             JAST.compilationUnit.reportSemanticError(line, "The member, "
636                     + member.name()
637                     + ", is not accessible because it's in a different "
638                     + "package.");
639             return false;
640         }
641     }
642 
643     /**
644      * Check the accesibility of a target type (from this type)
645      *
646      * @param line
647      *            the line in which the access occurs.
648      * @param targetType
649      *            the type being accessed.
650      * @return true if access is valid; false otherwise.
651      */
652 
653     public boolean checkAccess(int line, Type targetType) {
654         if (targetType.isPrimitive()) {
655             return true;
656         }
657         if (targetType.isArray()) {
658             return this.checkAccess(line, targetType.componentType());
659         }
660         return checkAccess(line, classRep, targetType.classRep);
661     }
662 
663     /**
664      * Check the accessibility of a type.
665      *
666      * @param line
667      *            the line in which the access occurs.
668      * @param referencingType
669      *            the type attempting the access.
670      * @param type
671      *            the type that we want to access.
672      * @return true if access is valid; false otherwise.
673      */
674 
675     public static boolean checkAccess(int line, Class referencingType,
676             Class type) {
677         java.lang.Package p1 = referencingType.getPackage();
678         java.lang.Package p2 = type.getPackage();
679         if (Modifier.isPublic(type.getModifiers())
680                 || (p1 == null ? "" : p1.getName()).equals((p2 == null ? ""
681                         : p2.getName()))) {
682             return true;
683         } else {
684             JAST.compilationUnit.reportSemanticError(line, "The type, "
685                     + type.getCanonicalName() + ", is not accessible from "
686                     + referencingType.getCanonicalName());
```

```
687             return false;
688         }
689     }
690
691     /**
692      * Resolve this type in the given context. Notice that this has meaning only
693      * for TypeName and ArrayTypeName, where names are replaced by real types.
694      * Names are looked up in the context.
695      *
696      * @param context
697      *            context in which the names are resolved.
698      * @return the resolved type.
699      */
700
701     public Type resolve(Context context) {
702         return this;
703     }
704
705     /**
706      * A helper for constructing method signatures for reporting unfound methods
707      * and constructors.
708      *
709      * @param name
710      *            the message or Type name.
711      * @param argTypes
712      *            the actual argument types.
713      * @return a printable signature.
714      */
715
716     public static String signatureOf(String name, Type[] argTypes) {
717         String signature = name + "(";
718         if (argTypes.length > 0) {
719             signature += argTypes[0].toString();
720             for (int i = 1; i < argTypes.length; i++) {
721                 signature += ", " + argTypes[i].toString();
722             }
723         }
724         signature += ")";
725         return signature;
726     }
727
728 }
729
730 /**
731  * Any reference type that can be denoted as a (possibly qualified) identifier.
732  * For now, this includes only class types.
733  */
734
735 class TypeName extends Type {
736
737     /**
738      * The line in which the identifier occurs in the source file.
739      */
740     private int line;
741
742     /** The identifier's name. */
743     private String name;
744
745     /**
746      * Construct an TypeName given its line number, and string spelling out its
747      * fully qualified name.
748      *
749      * @param line
750      *            the line in which the identifier occurs in the source file.
751      * @param name
752      *            fully qualified name for the identifier.
753      */
754
755     public TypeName(int line, String name) {
```

```
        this.line = line;
        this.name = name;
    }

    /**
     * Return the line in which the identifier occurs in the source file.
     *
     * @return the line number.
     */

    public int line() {
        return line;
    }

    /**
     * Return the JVM name for this (identifier) type.
     *
     * @return the JVM name.
     */

    public String jvmName() {
        return name.replace('.', '/');
    }

    /**
     * Return the JVM descriptor for this type.
     *
     * @return the descriptor.
     */

    public String toDescriptor() {
        return "L" + jvmName() + ";";
    }

    /**
     * Return the Java representation of this type. Eg, java.lang.String.
     *
     * @return the qualified name.
     */

    public String toString() {
        return name;
    }

    /**
     * Return the simple name for this type. Eg, String for java.lang.String.
     *
     * @return simple name.
     */

    public String simpleName() {
        return name.substring(name.lastIndexOf('.') + 1);
    }

    /**
     * Resolve this type in the given context. Notice that this has meaning only
     * for TypeName and ArrayTypeName, where names are replaced by real types.
     * Names are looked up in the context.
     *
     * @param context
     *            context in which the names are resolved.
     * @return the resolved type.
     */

    public Type resolve(Context context) {
        Type resolvedType = context.lookupType(name);
        if (resolvedType == null) {
            // Try loading a type with the give fullname
            try {
```

```
                resolvedType = typeFor(Class.forName(name));
                context.addType(line, resolvedType);
                // context.compilationUnitContext().addEntry(line,
                // resolvedType.toString(),
                // new TypeNameDefn(resolvedType));
            } catch (Exception e) {
                JAST.compilationUnit.reportSemanticError(line,
                        "Unable to locate a type named %s", name);
                resolvedType = Type.ANY;
            }
        }
        if (resolvedType != Type.ANY) {
            Type referencingType = ((JTypeDecl) (context.classContext
                    .definition())).thisType();
            Type.checkAccess(line, referencingType.classRep(), resolvedType
                    .classRep());
        }
        return resolvedType;
    }
}

/**
 * The (temporary) representation of an array's type. It is built by the Parser
 * to stand in for a Type until analyze(), at which point it is resolved to an
 * actual Type object (having a Class that identifies it).
 */

class ArrayTypeName extends Type {

    /** The array's base or component type. */
    private Type componentType;

    /**
     * Construct an array's type given its component type.
     *
     * @param componentType
     *            the type of its elements.
     */

    public ArrayTypeName(Type componentType) {
        this.componentType = componentType;
    }

    /**
     * Return the (component) type of its elements.
     *
     * @return the component type.
     */

    public Type componentType() {
        return componentType;
    }

    /**
     * Return the JVM descriptor for this type.
     *
     * @return the descriptor.
     */

    public String toDescriptor() {
        return "[" + componentType.toDescriptor();
    }

    /**
     * A string representation of the type in Java form.
     *
     * @return the representation in Java form.
     */

```

```
894     public String toString() {
895         return componentType.toString() + "[]";
896     }
897 
898     /**
899      * Resolve this type in the given context.
900      *
901      * @param context
902      *            context in which the names are resolved.
903      * @return the resolved type.
904      */
905 
906     public Type resolve(Context context) {
907         componentType = componentType.resolve(context);
908 
909         // The API forces us to make an instance and get its
910         // type.
911         Class classRep = Array.newInstance(componentType().classRep(), 0)
912                 .getClass();
913         return Type.typeFor(classRep);
914     }
915 
916 }
917 
```