Compound Types in Java

The aim of this project was to produce a working implementation of the extension of the java type system proposed by Büchi and Weck [1]. The two extensions described in that paper, compound types and aliases, have been implemented as an extension of the jaco compiler [4].

Büchi and Weck defined compound types as an anonymous reference type, defined by a set of interfaces and zero or one non-final class, called the constituent types. The compound type an extension of its constituent types. As such, every method and field defined in one of the constituent type is available in the compound type and have the same accessibility. A compound type does not define any additional method or field. If the set of types does not contain a class, the compound types is assumed to descend from `java.lang.Object`. A variable whose declared type is a compound type must have as its value either `null` or a reference to an object that inherits from the class, if one has been given, and that implements all the constituent interfaces. Compound types are meant to be used as a parameter type, a variable type, a return type, the return type methods, cast operator and operand of the instanceof operator.

This project also accepts compound types when they appear on a list of interface names, following `extend` in an interface declaration, or `interfaces` in a class declaration. In this case, it does not represent a reference type, but a series of interface type names.

CJava, the compiler extension implemented in this project, consists of a new syntactic and semantic analyzer that produces a tree containing both standard jaco nodes and the nodes added by cjava. The core of the cjava extension, the translator, later removes the new nodes to produce a semantically correct tree that can be parsed and converted into a java .class file. The main work of cjava is thus to translate cjava code into standard java code.

This report is divided in two parts. The first part contains the description of the additions made to the jaco compiler that were necessary to implement compound types. The second part describe the modifications made to implement alias types.

Naming Convention

In the examples, the names `C, C1, C2,` etc. denote a class, and `I, I1, I2,` etc., an interface.

Part 1: Compound Types

1.1 Language modifications

The java grammar has been modified to accept compound types wherever both instances and class names are legal. A compound type is a comma-separated list of reference types enclosed within brackets. Nesting is supported.

Compound types can contain zero or one non-final class and any number of interfaces. A compound containing only one interface or one class is equivalent to that class or interface. If no class type is part of the constituent type list, `java.lang.Object` is added to list. The order in which the types appear on

the type list is not significant.

For the compound type to be legal, it is necessary that all its constituent types be compatible. In particular, two constituent types may not declare a method with the same name that have different return types.

These are valid compound types : `[C, I1, I2]`, `[I1, I2]`, `[I]` (equivalent to `[Object, I]`, and `I`).

1.1.1 New Syntax

Here are modifications made to the standard java grammar, in EBNF form. It is based on the grammar defined in the java language specification [2]. Note that, although creation of arrays of compound types are accepted by this grammar, they are rejected by the semantic analyzer.

```
ADDITIONS:

  ClassOrInterfaceTypeNoCompound ::=
               ClassType
             | InterfaceType

  CompoundType ::=
               "[" CompoundList "]"

  CompoundList ::=
               ClassOrInterfaceType:n
             | CompoundList "," ClassOrInterfaceType

CHANGES:

  ClassOrInterfaceType ::=
               ClassOrInterfaceTypeNoCompound
             | CompoundType


  ClassDeclaration ::= [Modifiers] "class" Identifier
                       [ "extends" ClassType ]
                       [ "implements" InterfaceTypeList ]
                       ClassBody

  ArrayType ::=
        ( PrimitiveTypeType | Name | CompoundType ) Dims


  ClassInstanceCreationExpression ::=
                [ Primary "." ] "new" ClassOrInterfaceTypeNoCompound "(" ArgumentLi
                | "new" ClassOrInterfaceTypeNoCompound "(" ArgumentList ")"

  CastExpression ::=
              ...
             | "(" CompoundType [Dims] ")" UnaryExpressionNotPlusMinus

  ClassOrArray ::=
               Name
              | ArrayType
              | CompoundType
```

1.2 Translation

The translator receives a semantically correct tree with cjava extensions and transforms it into a tree composed of the nodes understood by jaco. The tree will go through a semantic analyzer accepting only standard java and is then compiled normally.

1.2.1 General case

The translator needs to convert compound types to standard java types in order to produce valid java code. For that, it selects one of the constituent types and replaces the compound type with this type. The algorithm for selecting the standard type from the type list is:

- if a class different from `java.lang.Object` is found, select that class to represent the compound type.
- otherwise, alphabetically sort the full name of the types in the compound and choose the last one.

For example, the standard type used to represent `[I1, I3, I2]` is `I3`, since I1, I2 and I3 are all interfaces, and the standard type for `[java.util.Vector, I1, I2]` is the object class `java.util.Vector`.

Prior to applying this algorithm, nested compound types are converted into a simple list of standard types, duplicates are removed as well as the `java.lang.Object` type. This means that the following types are all equivalent : `[I, [Object, I], I], [Object, I], [I], [[I]], I .`

1.2.2 Basic Operations

1.2.2.1 Variable Declaration

Compound types in declarations are simply converted into standard java types using the algorithm described above. The same holds true for array declarations.

Example: These variable declarations:

```
[I1, I2] i1i2;
[I1, I2, C1] i1i2c1;
[I1, C3] i1;
[Object, I2] i2;
[[I1], [Object, I3]] i1i3;
[I1, I2][] a1;
[I2][][] a2;
```

will be translated into:

```
I2 i1i2;
C1 i1i2c1;
C3 i1;
I2 i2;
```

```
I3 i1i3;
I2[] a1;
I2[][] a2;
```

1.2.2.2 Assignment

It is often necessary to add casts in assignment operations containing instances of compound types, since the full type of the variables will not be known by the standard java compiler.

Example:

```
public class C extends C1 implements I1, I2
{
    void dummy()
    {
        [I1, I2] a;
        [I1, I2, C1] b;
        [I1, I2, I3][] d;
        [I1, I2][] e;

        b = this;
        a = b;
        b = null;

        d = null;
        e = d;
    }
    Object[] getArray()
    {
      return new C[2];
    }
}
```

will be translated into:

```
public class C extends C1 implements I1, I2 {
   void dummy() {
      I2 a;
      C1 b;
      I3[] d;
      I2[] e;

      b = this;
      a = (I2)b;
      b = null;

      d = null;
      e = (I2[])d;
   }
}
```

1.2.2.3 Accessing Methods and Fields

When accessing methods and fields of an expression whose type is a compound, the expression first needs to be converted into the type it is used as. The translator keeps note of where the appropriate method or field was found and adds casts whenever necessary.

Getting the class object of a compound type using `.class` (as in `[I1, I2].class`) is not possible, since no `Class` object exists in this case. Expressions of this type are rejected by the grammar.

Example:

```
public class C extends C1 implements I1, I2
{
    [I1, I2] a;
    [I1, I2, C1] b;

    void dummy()
    {
        b = this;
        a = this;
        b.i1();
        b.i2();
        b.c1();
        a.i1();
        a.i2();
        int w = a.i1 + a.i2;
    }
}

interface I1
{
  void i1();
}

interface I2()
{
  void i2();
}

class C1
{
  void c1();
}
```

the class C will be translated into:

```
public class C extends C1 implements I1, I2 {
   I2 a;
   C1 b;
   void dummy() {
      b = this;
      a = this;
      ((I1)b).i1();
      ((I2)b).i2();
      b.c1();
      ((I1)a).i1();
      a.i2();
```

```
      int w = ((I1)a).i1 + a.i2;
   }
}
```

1.2.2.4 Cast

Casts to a compound type are converted into a series of casts to each constituent type. This ensures that the run-time type of the object will be checked, and that any `java.lang.ClassCastExpression` will be thrown at this point, and not later.

Example:

```
[I1, I2, I3] b = ([I1, I2, I3])a;
[I1, I2] b = ([I2, I1])a;
[I1, I2, I3][] b = ([I1, I2, I3][])a;
```

will be translated into:

```
I3 b = (I3)((I2)((I1)a));
I2 b = (I2)((I1)a);
I3[] b = (I3[])((I2[])((I1[])a));
```

1.2.2.5 Instanceof

Checking that a variable can be safely cast to a compound type is equivalent to checking that it implements or extends each of the constituent types. Therefore, the translator converts Instanceof operations on compound types into a series of standard instanceof operations.

To avoid duplicating side effects in a method call or breaking atomicity, a temporary local variable is used to store the result of the expression between instanceof operations unless the expression being tested is already a local variable.

Example: These variable declarations:

```
public class C
{
    [I1, I2] a;
    [I1, I2, C1] b;

    void dummy(Object o)
    {
        if( o instanceof [I1, I2])
            {
            }

        if(getObject() instanceof [I1, I2, C1])
            {
            }

        if(getObject() instanceof [I1, I2][])
```

```
            {
            }

        if(a instanceof [I1, I2, C1])
            {
            }
    }

    Object getObject()
    {
        return this;
    }
}
```

will be translated into:

```
public class C {
   I2 a;
   C1 b;

   void dummy(Object o) {
      java.lang.Object __cjava_t$0;
      if ( o instanceof I2 && o instanceof I1) {
      }

      if ( (__cjava_t$0 = getObject()) instanceof C1
           && __cjava_t$0 instanceof I2
           && __cjava_t$0 instanceof I1) {
      }

      if ( (__cjava_t$0 = getObject()) instanceof I2[]
           && __cjava_t$0 instanceof I1[]) {
      }

      if ( (__cjava_t$0 = a) instanceof C1
           && __cjava_t$0 instanceof I2
           && __cjava_t$0 instanceof I1) {
      }
   }
   Object getObject() {
      return this;
   }
   public C() {
      super();
   }
}
```

1.2.3 Exception Handling

Exception handling is normally done by the virtual machine, and not by the compiler. A standard java compiler needs only check which exceptions can be thrown and caught at a certain point in the source code. But in order to accept compound types in catch clauses, it is necessary to writes code that simulates what the JVM does for standard types.

The main idea of the algorithm that writes the exception handling code for compound types is to

transform a catch clause that is passed a compound type into a clause catching a standard class type and then checking what interfaces the class implements using instanceof operations:

```
catch([Exception, Serializable] e)
{
    ...
}
```

becomes

```
catch(Exception e) {
    if(e instanceof Serializable) {
        ...
    }
    else {
        throw e; // re-throw, will be handled later
    }
}
```

When more than one catch clause is present, re-throwing the exception if it does not implement the constituent interfaces is not always enough:

```
catch([Exception, Serializable] e)
{
    ...
}
catch(Exception e)
{
    ...
}
```

must becomes

```
catch(Exception e)
{
  if(e instanceof Serializable)
  {
     ... code for [Exception, Serializable]
  }
  else
  {
     ... code for Exception
  }
}
```

The algorithm tries to guess which clauses must be put together in that way so that the right exception handling code is executed.

1.2.3.1 Details of the algorithm

The algorithm is based the fact that catchable compound types always contain exactly one class, which is a subclass of Throwable, as well as some interfaces.

The original catch clauses that have to be put together in one catch clause by the translator form a `CatchFamily`. The choice of which family a clause must belong to is based on the possibility of an object matching the class of the compound, but not the interfaces, having to 'fall through' to another catch clause.

This is done by iterating over the array of catch clauses:

- As long as there are no families that may receive new members (they are said to be closed):
  - when a compound type is found, a new family is created that contains the compound type. The 'head' of the family, the real class that will be caught by the resulting catch clause, is set to the class found in the compound. The new family is open.
  - when a class type is found, no family is created. Or rather, a new closed family is created containing only this catch clause.

If there are open families, they are checked in turn to find one that may receive the new catch clause. It is the case when the class caught by the clause is a subclass or a superclass of the head of a family. If a family is found for the clause, it receives the new member, and if the clause caught a non-compound type, the family is closed. Closing at this time is OK, since the java compiler ensures that catch clauses appearing afterwards cannot catch a subtype of any of the family's class.

If no family were found for the new catch clause, it is handled as if there were no open families.

A case may need special handling: when a new catch clause is superclass of more than one head of open families, the families must be merged into one, with the class caught by the new clause as the head.

1.2.3.2 Complete example

```
import java.io.*;

interface A1
{
 void a1();
 Object obj();
}

interface A2
{
}

interface A3
{
}

class Test
{
    void boo() throws Throwable, IOException, EOFException
    {
    }
```

```
    void dummy() throws Throwable
    {
        try
            {
                boo();
            }
        catch([NullPointerException, A1] npea1)
            {
                System.out.println("npea1");
            }
        catch([EOFException, A2] iieo)
            {
                System.out.println("iioe");
            }
        catch(NumberFormatException nfe)
            {
                System.out.println("nfe");
            }
        catch(NullPointerException npe)
            {
                System.out.println("npe");
            }
        catch(IOException ioe)
            {
                System.out.println("ioe");
            }
        catch(RuntimeException rte)
            {
                System.out.println("rte");
            }
        catch([Exception, A1] ea1)
            {
                System.out.println("ea1");
            }

    }
}
```

will be translated into:

```
import java.io.*;

interface A1 {
   void a1();
   Object obj();
}

interface A2 {
}

interface A3 {
}

class Test {
   void boo() throws Throwable, IOException, EOFException {
   }

   void dummy() throws Throwable {
      java.lang.Object __cjava_t$0;
```

```
      try {
         boo();
      }  catch (NumberFormatException nfe) {

         System.out.println("nfe");                    // handle NumberFormatExcept

      } catch (java.lang.NullPointerException __cjava_t$1) {
         if (__cjava_t$1 instanceof A1) {
            java.lang.NullPointerException npea1 = __cjava_t$1;

            System.out.println("npea1");               // handle [NullPointerExcept

         } else {
            java.lang.NullPointerException npe = __cjava_t$1;

            System.out.println("npe");                 // handle NullPointerExcepti

         }

      } catch (java.io.IOException __cjava_t$1) {
         if (__cjava_t$1 instanceof java.io.EOFException && __cjava_t$1 instanceof A
            java.io.EOFException iieo = (java.io.EOFException)__cjava_t$1;

            System.out.println("iioe");               // handle [EOFException, A2]

         } else {
            java.io.IOException ioe = __cjava_t$1;

            System.out.println("ioe");                // handle IOException

         }

      } catch (RuntimeException rte) {

         System.out.println("rte");e                  // handle RuntimeException

      } catch (java.lang.Exception __cjava_t$1) {
         if (__cjava_t$1 instanceof A1) {
            java.lang.Exception ea1 = __cjava_t$1;

            System.out.println("ea1");                // handle [Exception, A1]

         } else
            throw __cjava_t$1;                        // rethrow exception if not ha
      }
   }
   Test() {
      super();
   }
}
```

Here, the code for handling `[NullPointerException, A1]` and the one for handling `NullPointerException` have been put together, so that if the exception implements A1, it is treated as a `[NullPointerException, A1]` Otherwise the code for handling `NullPointerException` is executed.

`[EOFException, A2]` and its superclass `IOException` are handled in the same way: `IOException` objects are caught and a check is made to see whether they are actually `[EOFException, A2]` or plain

`IOException`.

1.2.4 Class and Interface Declaration

The only place a compound type may be found in a class or interface declaration is on the list of interfaces implemented by a class or extended by an interface. Such compound types may only contain interfaces. The translator replaces the compound type with an enumeration of its constituent interfaces.

This behavior differs from Büchi and Weck's definition of compound types [1]; they didn't allow compound types in `implements` or `extends` clauses.

Example:

```
abstract public class C implements [I1, I3, I2] I4
{

}

public interface I extends [I2, I4, I1], I3
{

}
```

will be translated into:

```
abstract public class C implements I3, I2, I1, I4 {

}

public interface I extends I4, I2, I1, I3 {

}
```

1.2.5 Method Declaration

In method declarations, translating compound types into their standard types is not sufficient, because in some cases, it may be possible to overload a method taking a compound type as an argument with another one taking the standard type corresponding to the compound type. After the compound type has been translated into a standard type, the two methods would have the same name and signature.

To avoid this, the translator renames all methods taking compound type arguments. The cjava compiler keeps track of these name changes and modifies method calls wherever necessary.

Example: A very heavily overloaded `val()` method in a class.

```
public class C
{
    int val()
```

```
    {
        return 1;
    }
    int val(int base)
    {
        return base+val();
    }
    int val(Object o)
    {
        return o.hashCode();
    }
    int val(I1 i)
    {
        return i.i1*2 + val((Object)i);
    }
    int val(I2 i)
    {
        return i.i2*2 + val((Object)i);
    }
    int val([I1, I2] i)
    {
        return val((I1)i) + val((I2)i);
    }

    int val([I3, I1, I2] i)
    {
        return i.i3 + val(([I2, I1])i);
    }
}
```

will be translated into:

```
public class C {
   int val() {
      return 1;
   }
   int val(int base) {
      return base + val();
   }
   int val(Object o) {
      return o.hashCode();
   }
   int val(I1 i) {
      return i.i1 * 2 + val((Object)i);
   }
   int val(I2 i) {
      return i.i2 * 2 + val((Object)i);
   }
   int val$cj$XPLI1$LI2$$XI(I2 i) {
      return val((I1)i) + val((I2)i);
   }
   public C() {
      super();
   }

   int val$cj$XPLI1$LI2$LI3$$XI(I3 i) {
      return i.i3 + val$cj$XPLI1$LI2$$XI((I2)((I1)i));
   }
}
```

1.3 Class Attributes

When importing a class, the cjava compiler needs to be able to find out the real signature and name of the methods taking compound types as their argument, as in the example of the previous section. This is done using class attributes.

CJava defines two new attributes, `CJRealSig` and `CJRealVar`. `CJRealSig` saves the signature and name of the methods having compound types in their arguments, return value, or thrown exception list, and `CJRealVar`, the type of class variables.

The new attributes are ignored by the JVM, as defined in the JVM specifications [2]. Only the cjava compiler reads and interprets them.

An extended version of java signatures has been defined for these attributes. In addition to the standard types, the extended signature may contain compound types. The format used to convert compound types into strings is : `"P" { constituent-type-signature } ";"`. For example, the signature of `[Vector, I1]` is `PLjava/util/Vector;LI1;;`.

1.3.1 Class Variable Declarations

`CJRealVar` is a field attribute that contains the index in the pool of the extended signature of the field's type.

1.3.2 Method Declarations

`CJRealSig` is a method attribute.

It contains :

- the real name of the method, as the programmer would write it
- the extended signature of the method
- the number of exceptions declared to be thrown by the method
- the extended signature of these exceptions, if any

1.4 Limitations

While arrays of compounds are legal, the compiler refuses to create new ones, because element types must have a run-time type that actually implements the constituent interfaces and extends the constituent class. Otherwise the JVM throws an exception at some point. If arrays of compound types were created, a seemingly innocent conversion from one array to another would lead to an invalid cast operation that the JVM would not accept.

Example: In this code, the conversion from `[I1, I2][]` to `I1[]` should be valid, since `I1` is a supertype of `[I1, I2]`. Yet, the JVM cannot accept it, since what it sees is a conversion from an array of `I2` to an array of `I1`.

```
[I1, I2][] array = new [I1, I2](3);
I1[] array2 = array;
```

would have been be translated into:

```
I2[] array = new I2(3);
I1[] array2 = (I1[])array; // a ClassCastException is thrown when this line is execu
```

It would have been possible to create arrays of compounds but forbid conversion. It would however be even more limiting, since perfectly safe conversions would have then been impossible.

Part 2: Type aliases

Büchi and Weck [1] described another extension to the java type system that would make compound types simpler to use. The idea is to use an arbitrary name as a synonym of a class, interface or compound type, much in the way of C's `typedef`. Then, it would not always be necessary to write long compound type definitions.

2.1 Language Modifications

The grammar has been modified so that it is possible to declare aliases wherever class declaration is legal. That is, in a package, and within a class or method body. Alias declarations are of the form : modifiers ("class" | "interface") name "=" type ";". Only classes, interfaces and compound types may be aliased.

Some alias declarations:

```
public class AliasToVector = java.util.Vector;
interface AliasToList = java.util.List;
public class Alias1 = [C1, I1, I2];
```

An alias defined in a package may be public or package-local. In a class, it can also be private or protected. A local alias defined in the body of a method has the same scope as a variable defined at the same place in the code would have.

Defining an alias with a more relaxed protection than the class or interface it is aliased to is illegal. For example, it is not possible to define a public alias to a private inner class, because it would have the effect of making the inner class public.

2.1.1 New Syntax

The modifications that have been made to the standard java grammar, following those described in 1.1.1, EBNF form :

```
ADDITIONS:
```

```
   AliasDeclaration ::=
                 Modifiers ( "class" | "interface" ) Identifier "=" ClassOrInterfac

   LocalAliasDeclarationStatement ::= Modifiers ( "class" | "interface" ) LocalAlias

   LocalAliasDeclarations ::=
                 Identifier "=" ClassOrInterfaceType
               | LocalAliasDeclarations "," Identifier "=" ClassOrInterfaceType

CHANGES:

  TypeDeclaration ::=
                 ClassDeclaration
               | InterfaceDeclaration
               | AliasDeclaration


  BlockStatement ::=
                 LocalVariableDeclarationStatement
               | LocalAliasDeclarationStatement
               | Statement
               | ClassDeclaration

  ClassMemberDeclaration ::=
                 FieldDeclaration
               | MethodDeclaration
               | InnerClassDeclaration
               | AliasDeclaration
```

2.2 Translation

Aliases are always translated into the type they represent.

2.2.1 Alias Declaration

The translator converts alias declarations into declarations of classes with the same name and protection flags. Private and local aliase declarations can be removed from the tree, since they cannot be accessed from outside the current source file and are never needed at run-time.

Classes representing aliases are exported with a special attribute tagging them as alias definitions. To the virtual machine, they appear as empty classes.

Example:

```
public interface AliasToI1 = I1;
interface AliasToI1I2 = [I1, I2];

public class C
{
    private interface AliasToI2 = I2;
    public interface AliasToI1I2I3 = [I1, I2, I3];

    void dummy(Object o)
    {
        class LocalAlias = I1;
        LocalAlias a = (LocalAlias)o;
```

```
        a.i1();
    }
}
```

will be translated into:

```
public class AliasToI1 {
}

class AliasToI1I2 {
}

public class C {
    /* AliasToI2 declaration removed */

    static public class AliasToI1I2I3 {
    }

    void dummy(Object o) {
        I1 a = (I1)o;
        a.i1();
    }
}
```

2.3 Class Attributes

Classes representing aliases are tagged with the class attribute `CJAlias`. This attribute tells the cjava compiler which alias has been defined without it having to recompile the the source code.

The attribute contains an index to the extended signature of the aliased type in the pool. This signature supports compound types as well as aliases. Since aliases are indistinguishable from classes in the extended signature, the compiler can only tell the difference after parsing the class file.

2.4 Limitations

Since most alias definitions are saved in class files, the cjava compiler generates many files that are never used at run-time. Only the cjava compiler needs them. This has the drawback of mixing class files used at run-time with pure compiler constructs.

Bibliography

- M Büchi, W Weck. Java Needs Compound Types. TUCS Technical Report No. 182, June 1998. http://www.tucs.abo.fi/publications/techreports/TR182.html
- J. Gosling, B. Joy, G. Steele. The Java Language Specification. http://java.sun.com/docs/books/jls/html/index.html 1996
- T. Lindholm, F. Yellin. The Java(tm) Virtual Machine Specification, Second Edition. http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html. 1999
- M. Zenger. Erweiterbare Übersetzer. Diplomarbeit Universität Karlsruhe. 1998

Stephane Zermatten

Last modified: Wed Jun 21 14:55:19 CEST 2000