Proceedings of the Joint ACM Java Grande/ISCOPE Conference, Stanford, California, June, 2001

Kava: A Java Dialect with a Uniform Object Model for Lightweight Classes

David F. Bacon
IBM T.J. Watson Research Center

ABSTRACT

Object-oriented programming languages have always distinguished between "primitive" and "user-defined" data types, and in the case of languages like C++ and Java, the primitives are not even treated as objects, further fragmenting the programming model. The distinction is particularly problematic when a particular programming community requires primitive-level support for a new data type, as for complex, intervals, fixed-pointed numbers, and so on.

We present Kava, a design for a backward-compatible version of Java that solves the problem of programmable lightweight objects in a much more aggressive and uniform manner than previous proposals. In Kava, there are no primitive types; instead, object-oriented programming is provided down to the level of single bits, and types such as int can be explicitly programmed within the language. While the language maintains a uniform object reference semantics, efficiency is obtained by making heavy use of *unboxing* and *semantic expansion*.

We describe Kava as a dialect of the Java language, show how it can be used to define various primitive types, describe how it can be translated into Java, and compare it to other approaches to lightweight objects.

1. INTRODUCTION

The capability for programmers to add their own data types is a fundamental advantage of object-oriented programming. However, it has also created a gulf between the highly efficient primitive types like integers and the more flexible programmer-defined object types. The gulf has widened considerably in Java, where objects are inherently dynamically allocated and require substantial per-object space overhead.

While in some realms the lack of orthogonality between primitive types and objects is merely pedagogically unsatisfying or requires inconvenient extra programming, in other realms, notably numerical computing, it is of the very first importance. The inability of Java to provide high-performance complex numbers has led to substantial pressure to extend the language definition.

However, there are other types that would benefit from primitive-like efficiency as well: in the numerical community, interval arithmetic types, fixed-pointed numbers, and higher-precision arithmetic; in networking, IP or ethernet addresses; in graphics, point classes.

In this paper, we show how an object-oriented language can be defined without any primitive types at all, and yet achieve the same efficiency as languages that make use of primitive types. Furthermore, we show how this can be done in the context of Kava, a backward-compatible extension to Java that compiles into standard Java class files.

We do this by adding value and enum types to Java, and by making extensive use of *semantic expansion* [21], in which object types are replaced by primitive types at the intermediate-language level by the JIT compiler. Essentially, semantic expansion is the old compiler trick of recognizing certain library functions and treating them as built-in primitives. Therefore, translation of value and enum data types from source code to bytecode does not actually optimize for speed; instead, it optimizes for ease of semantic expansion because this will lead to order-of-magnitude speedups in the back-end.

In Kava, int is part of the package kava.lang.primitive, and is defined as a value object containing an array of 32 enumerations objects of type bit, which can have the value zero or one. There is nothing the programmer can do to observe that int is not a fully general object. And yet it can be implemented as a 32-bit register value.

By making the lightweight class facility fully general, Kava provides a number of advantages:

1. Programmers can add new lightweight types and achieve performance comparable with that of primitive data types;
2. The programming model is simplified because the distinction between primitives and objects has been removed; and
3. The language design is simplified and more easily verifiable because a larger amount of the language is in libraries, and there is no need for large numbers of rules for primitive types that must be included in the language specification and verified on an ad-hoc basis.

We have used Jikes [3] to prototype parts of an earlier version of Kava. We are beginning to implement Kava using a combination of front-end and virtual machine transformations with Jikes and the Jalapeño Java Virtual Machine [2]. We are presenting this work in an early form in the hopes of both getting useful feedback from the Java Grande community and of influencing the Java Grande proposals for lightweight classes.

The paper is organized as follows: Section 2 describes the Kava language by example, and shows how some of the predefined types of Java can be defined. Section 3 describes how Kava is translated into standard Java bytecodes and implemented efficiently in the run-time system. Section 4 discusses how well Kava meets the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Java Grande/ISCOPE '01 Stanford, CA, USA
Copyright © 2001 ACM 0-89791-88-6/97/05 $5.00.

requirements of the Java Grande community. Section 5 discusses related work, and is followed by our conclusions and plans for future work.

2. THE KAVA LANGUAGE

In this section we will describe Kava as an extension to the Java [7] programming language. We will describe the additional facilities, and show how they can be used together to define more complex types. Various subtleties must necessarily be omitted due to space constraints.

2.1 Enumerations

In order to build up the other "primitive" data types like integers, it is necessary to be able both to provide access to bit-level programming and to provide a way of iterating over collections of bits without requiring the existence of an integer type in the language. Enumerations fulfill both of these requirements.

Enumerations are of course familiar from many other programming languages. As will slowly become apparent, what is different in our approach is our ability to treat them like any other objects.

An *enumeration* is a special kind of class which enumerates its possible values. For instance, the declaration

```
enum color { red, green, blue }
```

defines an enumeration class with three values, named red, green, and blue. Enumeration identifiers are visible in the enclosing scope if there are no conflicts and they are not hidden by other variables. Thus it is usually possible to write

```
color c = red;
```

but it is always possible to name an enumeration value by its qualified name:

```
color d = color.blue;
```

Enumerations are ordered. They have no programmer-defined constructors, but the default constructor initializes an enumeration reference to the first value of the enumeration.

Enumerations are classes like any other, and may include methods, static variables, and so on. These are defined after the enumeration values, as for example

```
enum day {
  monday, tuesday, wednesday, thursday, friday,
  saturday, sunday;

  public boolean weekend() {
    return this == saturday || this == sunday;
  }
}
```

The only restriction on enumerations is that they may not have instance variables.

Enumerations also have a number of operators defined: the static operators first and last return the first and last values, respectively, in the enumeration. The prefix and suffix ++ and -- operators advance the operand to the next or previous value, respectively, like the analogous integer operators in Java. The auto-update operators wrap around, so that if day is sunday, ++day is equal to monday.

The toString() method returns the name of the enumeration value as a string and the hashCode() method returns the integer index of the enumeration value.

2.2 Operator Overloading

In Kava, there are no "primitive" types. Therefore, a general operator overloading facility is necessary so that the conventional prefix, postfix, and infix operators can be defined. Kava's operator overloading facility follows the same type rules as other method overloading.

Operators are defined using a syntax that mimics their use, with the keyword this standing in for the argument that represents the this operand in the use of the operator. The operator definition syntax is shown in the following operator for the day class shown above:

```
public day this + int that {
  int sum = (((int) this) + that) % 7;
  return day.fromInt(sum);
}
```

This code fragment defines a binary + operator which takes a day of the week and adds an integer number of days to it, returning the resulting day of the week.

Operator precedence is fixed, following the existing precedence conventions of Java.

2.3 Enumeration-Based Arrays

In Kava, Arrays can be indexed by either enumerations or integers. An array that is indexed by an enumeration is declared with the type of the enumeration that will be used as a subscript, as in:

```
int rating[color];
```

which defines rating as an array of three elements indexed by a value of the enumeration type color. The elements are initialized by applying the default (zero-argument) constructor for integers, so the array will be initialized to zeros.

Enumeration-indexed arrays never require subscript bounds checking, since the only possible subscripts are guaranteed to be members of the enumeration type.

The elements of the array are accessed or modified by using the subscripting operator, as in

```
int x = rating[red];
rating[blue]++;
```

Like integer-based arrays, enumeration-based arrays may have any number of dimensions. Initial values may also be specified using curly braces. For instance, the array composite tells whether a combination of colors is a composite color or not:

```
boolean composite[color][color] =
    { { false, true, true }, { true, false, true },
      { true, true, false } };
```

2.4 Putting it Together: Boolean

The three new language facilities we have described so far, namely enumerations, operator overloading, and enumeration-based arrays, are enough to define the "primitive" type boolean as shown in Figure 1.

Since a boolean has only two possible values and can not be subclassed, it can be represented without a class pointer or any other run-time information, unless it is passed as or assigned to a more general type. Therefore, boolean data types can be represented by a single bit.

Part of the details of how this works are due to the fact that enumerations are part of a more general facility in Kava for defining value types, which will be discussed below.

```
enum boolean {
   false, true;

   public boolean ! this {
      return not[this];
   }

   public static final boolean not[boolean] =
      { true, false };

   public boolean this & boolean that {
      return and[this][that];
   }

   public static final boolean and[boolean][boolean] =
      { { false, false }, { false, true } };

   // Similarly for other operators....
}
```

Figure 1: Use of enumerations, operator overloading, and arrays of enumerations to define the boolean **type.**

```
final value complex {
   public double re;
   public double im;

   public complex(double r, double i) {
      re = r; im = i;
   }

   public complex this + complex that {
      return new complex(re+that.re, im+that.im);
   }

   public complex this * complex that {
      double r = re*that.re - im*that.im;
      double i = re*that.im + im*that.re;
      return new complex(r,i);
   }

   // Similarly for other operators....
}
```

Figure 2: Partial definition of value class complex **in Kava**

We have shown the definitions for only two operations on boolean values: logical negation and boolean conjunction. The negation operator, a unary prefix operator, returns not[this]. The one-dimensional, enumeration-indexed array not is a truth-table for the boolean negation operator.

Similarly, the boolean and operator is defined using a two-dimensional boolean-indexed array, indexed by this and the parameter that.

The other operators are defined in a similar manner — and that's all there is to it!

2.5 Values

Values are first-class immutable objects.

When they are small, values can be stored in a register; when they are large, they can be copied by reference. This makes operations on values extremely efficient, and programmers should strive to maximize their use of values.

All instance variables of values must themselves be values or arrays of values. The instance variables of a value are all implicitly final.

Values may only be distinguished by their contents, not by their identity. The == operator, when applied to values, returns true if and only if the contained values are equal. The hashCode() method returns an integer that depends solely upon the contained variables.

Since values have no storage identity or mutability, they may not be synchronized and may not have finalizer methods. The synchronization methods wait(), notify(), and notifyAll() do not apply to values, and the synchronized keyword may not be applied to value methods.

All values are instances of the abstract class Value.

Enumerations are considered a special kind of value, and are instances of an abstract class Enum which is a subclass of Value. The relationship between enumerations, values, and objects is explained in detail in Section 2.7.

Values may be subclassed, but will generally not be optimized as highly if they are not final.

A good example of a value class is the complex class as shown in Figure 2.

Values may never be observed in a partially initialized state. To this end, the this pointer and the instance variables of a value may not be stored or passed as a parameter from a value constructor (if it is necessary to register the newly created value with a directory, an initializer block may used, but the value may not be changed within the initializer block).

On multiprocessors that are not sequentially consistent, the implementation may need to add synchronization at the end of the constructor to ensure that values are never observed in a partially initialized state.

Since there is no way to distinguish two references to equal values from two references to the identical value in storage, the implementation has a great deal of flexibility in optimization. For large value classes, interning techniques can be used to ensure that there is never more than one instance of a given value in the system.

Small final value classes that fit in registers can be passed in an "unboxed" format, since they do not need to carry their class pointer. The complex class defined above can be passed in two 64-bit floating point registers.

Use of unboxed final value classes is a fundamental optimization. It allows the removal of memory indirection in accessing the fields of a value, and means that composite values can be stored inline in objects.

A value that is declared as a final class can be implemented without a class pointer, provided that the value is "boxed" when it is assigned to a superclass.

Integers and other scalar types are defined as final value classes. Since they require no synchronization information or class pointer, they can be stored in a single word which contains only the 32 bits of the integer type.

2.5.1 Constructor Functions

Operators on values typically return a new value as a result. But since the fields of a value can only be modified in a constructor, this means that there may be many constructors with the same type signature. Therefore, Kava allows for explicitly named constructors.

For example, we might want two different complex constructors that each take only one double value, in which case we could add these two constructors to the class defined above:

```
public complex(double r) {
   re = r; im = 0.0;
}

public complex..imaginary(double i) {
   re = 0.0; im = i;
}
```

The second form is a *named constructor*, and operates exactly like other constructors. It is invoked as follows:

```
complex c = new complex..imaginary(1.0);
```

The double-dot notation serves to distinguish named constructors from inner class constructors.

2.6 Putting it Together: Unsigned Byte

Figure 3 shows how enumerations, values, and constructor functions can be combined to define an 8-bit unsigned integer data type, ubyte.

We assume that a bit data type has already been defined; like boolean it is a two-valued enumeration, but in addition to logic operators it also supports relational and arithmetic operators.

We begin by defining byteindex, an enumeration type with eight values that is used to index the bits of the byte.

The ubyte type is a final value, indicating that it will not change after initialization and that the type will not be subclassed.

The data field of ubyte is an enumeration-indexed, fixed-size array. This allows the data field to by stored inline in the object, and since it is a final value, there is no need for a class pointer or any synchronization information. Therefore, the object can be represented inline as an eight-bit object (unless it is cast to Value or Instance, in which case it is boxed).

The less-than operator shows how a boolean operator is defined: it iterates over the bits of the byte, from highest to lowest, applying the bit-wise less-than operator.

The complement operator returns a new value which is the complement of the unary argument; since values may not be modified once they have been created, it uses the named constructor function complement to build the new value and return it. The bit-wise and operator is similar except that it takes two arguments.

The binary addition operator performs unsigned addition using the sum() and carry() functions of the bit class. In hardware terms, this is a description of a ripple-carry adder, and could be synthesized as such with a design compiler capable of recognizing the idiom.

Note that the entire ubyte data type has been defined without any reference to int. It only uses previously defined enumeration types, and arrays of enumeration types, and the ability to iterate over those arrays.

2.6.1 Iteration

Also note that we have use do-while iteration rather than for loops. The reason is that Java's for loops, derived from C, rely on there being a value outside of the enumeration range — at the end of a typical Java for loop, the iteration variable is one past the final value that was used inside the loop. Since we wish to iterate over the entire set of values in the enumeration, there is no such value that is "one past".

We considered some alternatives, such as adding IEEE floating point not-a-number (NaN) style values [4] to enumerations. This would solve the iteration problem, but would introduce a larger problem because the number of bits required to represent bi-valued enumerations like bit and boolean would double. While not a prob-

```
enum byteindex { b0,b1,b2,b3,b4,b5,b6,b7 };

final value ubyte {

   private bit data[byteindex];

   public boolean this < ubyte b {
      byteindex x = byteindex.last;
      do {
         if (data[x] != b.data[x])
            return data[x] < b.data[x];
      } while (x-- != byteindex.first);
      return false;
   }

   public ubyte ~ this {
      return new ubyte..complement(this);
   }

   private ubyte..complement(ubyte b) {
      byteindex x;
      do {
         data[x] = ~ b.data[x];
      } while (x++ != byteindex.last);
   }

   public ubyte this + ubyte that {
      return new ubyte..sum(this,that);
   }

   private ubyte..sum(ubyte a, ubyte b)
   {
      bit c;

      byteindex x;
      do {
         data[x] = c.sum(a.data[x], b.data[x]);
         c = c.carry(a.data[x], b.data[x]);
      } while (x++ != byteindex.last);

   }

   public ubyte this >> byteindex s {
      return new ubyte..shiftr(this, s);
   }

   private ubyte..shiftr(ubyte a, byteindex s) {
      byteindex x2 = byteindex.first;
      do {
         data[x2++] = a.data[s];
      } while (s++ != byteindex.last);
   }

   // Similarly for other operators....
}
```

Figure 3: Definition of an unsigned integer type that uses only enumerations.

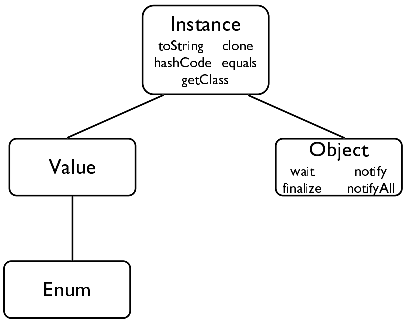

Figure 4: The top of the class hierarchy in Kava.

lem for scalars, this would certainly be problematic for arrays, especially when used to represent numeric data types like ubyte and int.

2.7 The Class Hierarchy

As we have seen, in Kava there are three kinds of classes: objects, values, and enumerations. Objects are essentially the same as objects in Java. Values are read-only class instances whose fields must themselves be values or arrays of values and may not be modified outside of the class constructor. Enumerations are a special kind of value whose domain is explicitly enumerated by the programmer.

These three kinds of objects are related in an inheritance hierarchy as shown in Figure 4. Instance, Value, and Enum are all abstract classes. Object is a concrete class for compatibility with Java. Enumerations implicitly extend Enum, values implicitly extend Value, classes implicitly extend Object.

Interfaces by default extend Instance. However, interfaces may extend any of these four abstract classes, in which case the interface is restricted to classes that implement the specified class type. This is in contrast to Java, where interfaces only extend interfaces.

A class or interface may not implement both Object and Value.

2.8 Bound and Optional References

In Java, an object variable can either be a reference to an object or null. With Kava, since we wish to have a uniform object semantics, and since we wish to be able to implement small objects like boolean efficiently, we can not admit the possibility of a null value for every object, since it would require at least an additional bit.

Therefore, we borrow from C++ the notion of two different types of references, one of which is always known to refer to an object, and one of which may refer to an object or be null. The former are called *bound references*, while the latter are called *optional references*.

For example, the statement

```
foo x;
```

defines x to be a *bound reference* to an object of type foo. Any attempt to assign null to x is either a compile- or run-time error.

For variables other than local variables (that require definite assignment), the above declaration is equivalent to initialization with the zero-argument constructor, as in:

```
foo x = new foo();
```

An *optional reference* denotes either an instance of a class or null. An optional reference is declared with the @ modifier, as in

```
@foo x = null;
x = new foo();
```

The default value for an optional reference is null. Assignment-compatible references, bound and optional, may be assigned to each other. If an optional reference is assigned to a bound reference, a check is inserted to ensure that it is not null, and if it is, a NullPointerException is thrown.

Unlike C++, in Kava objects are always heap-allocated by default, unless the compiler can determine that this is not necessary. Of course, the most important instance where heap allocation is not necessary is when the object is a final value or enumeration type, which can be stored inline in the object.

Because of the semantics of references, the declaration

```
int x;
```

declares x as a bound reference to an instance of type int. The null constructor is implicitly invoked, which returns the integer that represents 0. Therefore, integers have a semantics that is consistent with the object model of the language, but can be represented in a space efficient manner. If the compiler implements the operations on integers like + using the native instructions of the underlying machine, the same run-time efficiency is obtained as well.

2.8.1 Compatibility Issues

Ideally in Kava, all variables would be bound references by default. However, this is incompatible with Java, in which primitives always behave like bound references and user-defined classes are always optional references. For compatibility, we adopt a non-uniform approach in which values are bound references by default but can be explicitly specified to be optional, and vice-versa for all other types.

The modifier bound is provided in order to specify that a non-value variable is a bound reference rather than an optional reference. In the long term, this modifier can be deprecated and eventually removed, and programmers can migrate to a uniform syntax.

2.9 More on Operator Overloading

So far we have only described operator overloading in its basic form. However, additional features are required to allow the creation of types with the richness of Java's primitive types. These features are described here.

2.9.1 Cast Operators

Kava includes both implicit and explicit user-defined cast operators, as shown in these definitions from the class ubyte:

```
public (bit) this { return data[b0]; }

public implicit (uint) this {
   bit ival[wordindex];
   for (wordindex w = w0, byteindex b = b0; +b)
      ival[w++] = data[b];
   return new int(ival);
}
```

In this example, conversion from ubyte to bit is a narrowing conversion, so it has been defined as an explicit cast (the default), while the conversion from ubyte to uint is a widening conversion in which no information is lost, so it is declared as an implicit cast.

Implicit casts are restricted as follows: the compiler will only invoke a single implicit cast for a given expression; and both programmer-defined implicit casts and the implicit casts made possible by casting up the class hierarchy must impose a DAG structure on the class hierarchy.

The rules for method invocation resolution are unchanged, except that method invocation conversion is defined to be either widening reference conversion or implicit casting, instead of widening reference conversion or widening primitive conversion.

2.9.2 Assignment Operators

Normally, operators return a result and optionally update their operands. But when operators are applied to values, the values themselves can not be modified. For instance if x=1, then ++x does not change the value of 1, but changes x so that its new value is 2.

Assignment operators allow the reference of the operand to be replaced by the result of the operator. For instance, the prefix ++ operator of type int is declared as follows:

```
public assignment int ++ this {
   return this + 1;
}
```

Because ++ is an assignment operator, the result, which is a reference to an int, replaces the reference contained in the operand. An operand of an assignment operator must therefore be either a field of an object or a stack variable, or an element of an array.

Thus the first two expressions below are legal but the third is not:

```
int x = ++a;
int y = ++ p.foo().elts[2];
int z = ++(x+1);
```

Assignment operators include "=" and the compound assignment operators like "+=".

Most assignment operators return the same value that is assigned to the variable to which they are applied. However, the postfix increment and decrement operators are different. The value of of these operators is always the old value of the variable; the operator side-effects the variable.

A special type of assignment operator is provided to allow definition of these operators. Since the value of the expression is always the value of the object, there is no return statement and no return type is allowed to be defined. Instead, the assign statement is used to update the value of the variable:

```
public assignment this ++ {
   assign this + 1;
}
public assignment this -- {
   assign this - 1;
}
```

2.9.3 Additional Operators

Because Java does not have generalized l-values, the operator [ ]= is provided so that writers of array-like classes can provide modifier as well as accessor functions, as in the following methods from a StringBuffer class:

```
char this [ int i ] { return array[i]; }
char this [ int i ] = char c {
   array[i] = c;
   return c;
}
```

This requires special support from the parser to resolve ambiguity between the use of the subscript operator followed by the assignment operator, and the subscript/assignment operator.

We do not currently have a good solution to defining compound overloaded operators like "+=" for array updates. This may simply be a limitation of a language without l-value expressions.

The infix operator "[ ]", which takes one or more comma-separated arguments, is provided to support the creation of new scalar types that are composites of existing types. For instance, the statements

```
complex c = [3.5, 2.5];
ipaddress loopback = [classA, 127, 0, 0, 1];
```

initialize a complex value consisting of two doubles, and an ipaddress value consisting of four bytes.

2.10 Value Arrays

So far, the arrays we have shown that are elements of values are both private and only accessed locally. However, such arrays may be public, or may be passed to functions. In that event, since they are components of a value, they may not be modified. *Value arrays* allow such arrays to be accessed and operated upon in a limited (read-only) fashion.

A value array is declared with the modifier value, and can only be assigned from a static initializer or from another value array of the same type, or from an array which is a constructed instance variable of a value.

Here is an illustration of legal and illegal uses of value arrays:

```
value pa {
   int a[ ] = { 0, 1 };
}

pa pv = new pa();
value int[ ] x = pa.a;
int i = x[0];
x[0] = 1;          // illegal
Instance o = x;   // illegal
```

The array element pa.a can be assigned to the value array x, and can be accessed, but assigning to the value array or casting it to an Instance (or any type that is not a value array) is not allowed.

Note that array elements of values are not value arrays until the value has been constructed, because during that time the array may change. The general restrictions on passing references to partially constructed values (see Section 2.5) prevent the array from being passed as a parameter from within the constructor.

Value arrays and standard arrays are mutually incompatible and must be explicitly converted. A standard array may not be assigned to a value array because a value array includes a promise that its component values are immutable; a value array may not be assigned to a standard array because this would allow the elements of the value array to be modified. For every array class there are conversion functions:

```
int [ ] x = { 0, 1 };
value int[ ] y = x.toValue();
int[ ] z = y.toMutable();
```

These conversion functions should be applied sparingly, because they copy the array.

Note that value arrays are more restricted than const arrays in C++, since C++ allows arrays to be cast from type int[] to type const int[]. This merely restricts the operations that can be performed via the const reference to the array; but the array could continue to change via operations performed on the non-const reference to the array. Such a semantics is not acceptable in Kava, since it is not a pure value semantics.

3. COMPILING KAVA PROGRAMS

We have prototyped portions of Kava in Jikes [3], a Java-to-bytecode compiler developed at IBM, and are now beginning to build a complete system based on the language we have defined in this paper, which is much more compatible with Java than our original design. This system will combine front-end translation in Jikes and back-end semantic expansion in the Jalapeño Java Virtual Machine [2].

Our basic approach is that we translate values as Java classes and concentrate on making it easy for the JIT compiler to recognize idiomatic uses of classes. We then rely on semantic expansion [21] to convert the appropriate classes into register-level quantities.

At a minimum, semantic expansion is performed for all enum types and for all types in kava.lang.primitive, which includes definitions for all of the Java primitive types.

The advantage of our approach is that it does not require any changes to the virtual machine definition; therefore, Kava programs can be used experimentally (albeit slowly) on standard JVMs, and developers can continue to use the very rich set of development environments and tools that manipulate Java bytecodes.

The disadvantages are that without any semantic expansion, performance will be unacceptably slow, semantic expansion in the JIT may prove expensive, and removing primitive types may introduce significant amounts of extra lookup overhead into the compilation process.

We believe that these issues can be addressed. In this section we will describe our approach. Ultimately, the test will be in the implementation.

3.1 Source to Bytecode Compilation

The first step in implementing Kava is to modify a source-to-bytecode compiler (like `javac`) to recognize the Kava syntax and perform the appropriate translations. The main tasks are (1) expansion of enum and value types, (2) operator overloading, and (3) performing or inserting additional checks.

Enumerations are represented as final classes with a single private int instance variable. Static instances are created for each of the named enumeration values. Rather than go to elaborate lengths to avoid storage allocation, the code for operators like ++ is simple and uses new to allocate the successor object. It is left to the semantic expansion pass to remove the use of dynamic allocation.

Values are represented as classes whose instance variables are all final. The bytecode compiler generates methods that implement a value-based semantics for "==", "!=", hashCode(), and clone(). It checks that all instance variables are values or arrays of values, that the synchronized attribute is not applied to any of the methods, and that the this pointer and instance variables are not made visible within the constructor.

Operator overloading is performed with a straightforward mapping from operator syntax at the source level to method call expressions at the bytecode level. The major difference is in accommodating implicit casts, which replace the hardwired primitive widening conversions of Java. However, because the combined implicit reference conversions and implicit casts must still form a DAG relationship, the same rules are used to resolve ambiguities.

A variety of static safety checks are performed, including: that enumeration-based arrays are subscripted only by variables of the appropriate enumeration type; that null is not assigned to references; that assignment operators are not applied to elements of value arrays; and that synchronization and address examination operations are not applied to value types.

Safety checks that can not be performed at compile-time are also added, including: that synchronization and address examination operations are not performed on interfaces or Instance variables, and that pointers assigned to references are not null.

Some safety information must be encoded into class files. In particular: the index type of enumeration-indexed arrays, and the value and reference attributes of declarations. All other safety checks should be handled implicitly by the Java type system.

3.2 Front-End Semantic Expansion

Since Kava defines the integer plus operation as an iteration of bit-wise sums over two arrays of integer bit objects, it would be *extremely slow* on virtual machines that do not perform the appropriate semantic expansion. There is a tradeoff between a uniform approach which is secure because it relies on Java's type system for enforcement, and a less secure approach that performs more semantic expansion at the class file level, removing work from the JIT compiler.

Our plan is to translate classes from `kava.lang.primitive` into their Java primitive type equivalents during bytecode compilation, but to leave the rest of the semantic expansion process to the JIT compiler. This will allow legacy Java code to run efficiently, but will still provide strong typing.

3.3 Back-End Semantic Expansion

The rest of the semantic expansion is performed by the back-end of the Java compiler, usually the JIT, at the level of bytecode to intermediate-instruction translation.

The JIT compiler must perform the semantic expansion that makes enumerations and small, final value classes efficient. When a value class is loaded into the virtual machine, the JIT makes a decision as to whether to represent it as an unboxed value (all enumerations are represented unboxed).

For unboxed values, the class methods are converted to static methods that take unboxed parameters, and boxing and unboxing operations are added. Calls to the boxing and unboxing operations are introduced at any up- or down-casts from or to the value class.

For values it is often necessary to perform multiple levels of unboxing. For instance, to fully unbox the ubyte type from Figure 3, the JIT must examine the instance variable, which is an array of (unboxed) enumerations. The enumerations can be represented using one bit each, so the array can be represented using a byte variable in the JIT intermediate language. Since this is the only instance variable, the unboxed ubyte object also occupies one byte.

The JIT compiler must also convert the operations on the multi-level unboxed objects. In the case of ubyte, this means converting array element access and update into the appropriate shift-and-mask operations, and performing a boxing operation if the entire array is returned as the value of an expression.

Multi-level unboxing must also be performed when a value has multiple instance variables, in which case the JIT tries to pack the instance variables into as few bytes, shorts, words, or double-words as possible, while obeying the natural alignment of the component objects.

4. JAVA GRANDE REQUIREMENTS

The Java Grande Numerics Working Group report [15] lists five major areas that require attention in order for Java to become the language of choice for numerical computing. In this section, we examine how Kava meets these various requirements.

The five areas are (1) complex arithmetic, (2) lightweight classes, (3) operator overloading, (4) use of floating point hardware, and (5) multidimensional arrays.

Kava addresses the second and third requirements directly by adding lightweight classes and operator overloading to the language. Complex arithmetic can then be added as a lightweight class with overloaded operators, and can be compiled efficiently using the same semantic expansion techniques for complex numbers used by Wu et al [21]. However, greater optimization is possible in Kava because we can represent the target of a complex assignment in its semantically expanded form, rather than as an object.

Dense multidimensional arrays are not directly addressed by Kava, although the operator overloading facility does make it possible to implement them with one-dimensional Java arrays and explicit subscript computation.

Use of floating point hardware has already been addressed by the introduction of strictfp classes and methods. While it achieves the requirements, this solution is inelegant. Kava provides the potential for an alternate approach, based on the importing of different floating-point libraries. However, we have not explored this approach in detail.

4.1 Lightweight Classes

The main contribution of Kava is its unique and fully general way of providing lightweight classes (enumerations and values). We now compare the Kava approach to lightweight classes to the corresponding requirements of the Java Grande NWG report.

As recommended by the report, we adopt an approach in which Kava is translated into standard Java class files; no changes are required in the definition of the Java Virtual Machine [7].

However, Kava's values are more flexible than the NWG lightweight classes: in particular, Kava value classes can

- be either final or non-final,
- have user-supplied default constructors,
- invoke super from their constructors,
- implement interfaces,
- be null (if declared with the "@" optional reference modifier),
- be cast to interfaces or to Instance or Value,
- have the instanceof operator applied to them.

With the exception of the first three items, these properties apply to Kava enum types as well.

Kava is defined so that the program can never tell the difference between a deep and a shallow copy of a value; therefore, the implementation can choose whichever implementation of "=" is best suited. For small objects, the implementation should use deep copy; for large objects, a pointer copy.

4.2 Storage Issues

For enum and final value classes, there is no need to store a class pointer in the object unless they are upcast, in which case the runtime system must apply boxing transformations.

Once again, because Kava defines a pure semantics for values, they can always be stack-allocated. For values whose assignment operator is implemented by deep copy, escape analysis is not required when they are passed as parameters or returned as results. For values whose assignment operator is implemented by pointer copy, if they are stack-allocated then they must be deep-copied when they are assigned to non-local variables. Escape analysis can then be used to reduce or eliminate these deep copies.

4.3 Operator Overloading

Since there are no primitive types in Kava, operator overloading is a requirement rather than a desirable feature. Kava includes more overloading features (assignment operators, implicit and explicit casts) than would otherwise be the case because it has to be able to fully support the functionality of primitive types.

Kava's overloading syntax is novel. It is designed so that the definition syntax matches the invocation syntax, and avoids problems associated with multiple uses of a single operator (notably ++ and --). However, this is merely a syntactic choice and can easily be changed if it is unpalatable to the wider user community.

Because the primitive types of Java are simply value types in Kava, it is possible to define a + operator that applies to a double and a complex and returns a complex by making it an operator method of double. While this does not solve the problem of how to dynamically add symmetric operators to previously existing types, it is consistent with the way Java's type rules are currently defined.

A more general solution, like multi-methods (or reverse operators as suggested by Gosling [6]) could be added, but we have attempted to minimize language changes.

5. RELATED WORK

We have already compared Kava to related proposals for adding lightweight objects to Java in Section 4, and described how our work builds upon the work of Wu et al [21] for semantic expansion of complex types added as standard classes to Java.

Titanium [22] includes immutable classes share many of the positive attributes of Kava value classes. However, they are limited in a number of important respects: they are implicitly final, are not part of the class hierarchy, and in pathological cases non-value semantics can be observed.

A number of other approaches to adding complex numbers to Java exist [1, 17] but they are much narrower in scope than Kava.

Leroy [9] describes how unboxing can be applied to ML for floating point numbers and small tuples and records. His work is more general in that he applies unboxing to polymorphic code, while we require that data types be non-polymorphic in order to unbox them. His work is less general in that he only addresses the unboxing optimization. In more recent work, Leroy [10] suggests that local unboxing based on dataflow analysis and the creation of inlined function preludes and postludes to perform non-local unboxing can be as efficient as the type-based unboxing described in his previous work.

Other work on unboxing also explores various tradeoffs in the design and implementation space [14, 16, 18, 20].

With object-oriented programming languages, there has always been a distinction between "primitive" or "built-in" and user-defined object types. Although every data type in Smalltalk [5] was an object, there were special representations (like tagged integers) and it was not possible to define new object types with the same efficiency.

Eiffel [11] provided an `EXPANDED` declaration modifier which allowed the programmer to specify that a contained object should be stored directly in the object or stack frame, rather than included by reference. In this manner, variants of class `INTEGER` could be defined with their own operations, which only required one word of storage. However, Eiffel still is defined in terms of the familiar primitive types and does not have the facilities for building up arbitrary primitive-scale types as in Kava.

This project began as an outgrowth of our work with the Java-Time group at U.C. Berkeley [23]. The goals of that project were to enable the use of Java as a hardware design language using successive refinement and (at lower levels) automatic synthesis. We began with a desire to provide the same bit-level manipulation capabilities of hardware design languages like VHDL [8] and Verilog [13], while providing object-oriented facilities of abstraction and inheritance.

These goals are the same ones that have motivated the design and implementation of System C [19] by an industry consortium. System C is designed as a set of libraries for C++, and makes heavy use of operator overloading. The advantage of this approach is that language compatibility with C++ is retained; the disadvantage is that the features are neither as elegant nor as efficient as they could be with Kava.

6. CONCLUSIONS

We have described Kava, a backward-compatible dialect of Java that provides fully general lightweight classes. The Kava object model is completely uniform; from the programmers point of view, all types are objects. Primitive types can be defined within Kava, allowing far more of the language semantics to be mechanically encoded in a verifiable form.

Kava can be executed efficiently because the compiler or JIT can treat the objects in a non-uniform manner that is invisible to the programmer. The bytecode compiler translates Kava programs in a manner that makes this transformation simple for the JIT to perform.

Kava makes it easy to add *new* highly efficient data types to the language at the library level. We have shown that Kava meets many of the requirements of the Java Grande Numerics Working Group requirements in a much more general way than previous proposals. We are working on an implementation and hope to have code available for evaluation soon.

We have also shown how *existing* primitive types, like boolean and int can be defined in Kava. Does this mean we are suggesting that Kava programs should be interpreting integer addition by operating on enumerations down to the bit level? Of course not. Kava-aware JVMs will replace an integer addition method call with an invocation of a machine-level 32-bit register addition.

So why define the primitive types in Kava? In the short term, it allows us to move more of the language specification into an architecture-neutral executable format. If there is any doubt as to whether some operation is implemented correctly, the results using the semantically expanded code that calls native instructions can be compared with the results of bit-level interpretation.

In the longer term, the executable specification of primitives opens up new research areas. Can they be coupled with a description of the operations of the available hardware instruction set, for which a compiler-compiler then generates the translation to machine instructions automatically? Even more exciting, could the executable specifications of primitives be synthesized directly into hardware or firmware, allowing dynamic modification of the hardware instruction set? As more and more computation takes place on embedded CPUs and co-processors, we believe attacking such problems will become increasingly important.

Acknowledgments

Thanks to Richard Newton for inspiring me to start this work, and to Phillipe Charles, Joe Darcy, Deepak Goyal, William Kahan, Carlos Varela, John Vlissides, and the anonymous referees for helpful feedback.

7. REFERENCES

[1] BROPHY, J. Complex numbers in Java, 1999. Poster presentation, ACM Java Grande Conference.

[2] BURKE, M., ET AL. The Jalapeño dynamic optimizing compiler for Java. In *ACM Java Grande Conference* (San Francisco, CA, June 1999).

[3] CHARLES, P., AND SHIELDS, D. The Jikes project, 1999. Available at `oss.software.ibm.com/-developerworks/opensource/jikes/project`.

[4] CODY, W. J., ET AL. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro 4*, 4 (1984), 86–100.

[5] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachussetts, 1983.

[6] GOSLING, J. The evolution of numerical computing in Java, 1998. Available at `java.sun.com/-people/jag/FP.html`.

[7] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification*, second ed. The Java Series. Addision-Wesley, 2000. Available at `java.sun.com/docs/books/jls`.

[8] *IEEE Standard VHDL Language Reference Manual: IEEE Standard 1076-1993*, revised ed., 1994.

[9] LEROY, X. Unboxed objects and polymorphic subtyping. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages* (Jan. 1992), ACM Press, New York, New York, pp. 177–188.

[10] LEROY, X. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation* (June 1997), Technical Report BCCS-97-03, Computer Science Department, Boston College.

[11] MEYER, B. *Eiffel: The Language*. Prentice Hall, 2000.

[12] MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML — Revised*. MIT Press, 1997.

[13] MOORBY, P. R., AND THOMAS, D. E. *The Verilog Hardware Description Language*, fourth ed. Kluwer Academic, 1998.

[14] MORRISON, R., DEARLE, A., CONNOR, R. C. H., AND BROWN, A. L. An ad-hoc approach to the implementation of polymorphism. *ACM Trans. Program. Lang. Syst. 13*, 3 (1991), 342–371.

[15] NUMERICS WORKING GROUP, JAVA GRANDE FORUM. Improving Java for numerical computation, 1998. Available at `math.nist.gov/javanumerics/-reports/jgfnwg-01.html`.

[16] PEYTON JONES, S. L., AND LAUNCHBURY, J. Unboxed values as first class citizens. In *Functional Programming Languages and Computer Architecture: 5th ACM Conference* (Cambridge, Massachussetts, Aug. 1991), J. Hughes, Ed., vol. 523 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 636–666.

[17] PHILIPPSEN, M., AND EDWIN GÜNTHNER. cj: A new approach for the efficient use of complex numbers in Java, 1999. Available at `wwwipd.ira.uka.de/~gunthner`.

[18] SHAO, Z. Flexible representation analysis. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997), pp. 85–98.

[19] *System C Version 1.1 User's Guide*, 2000. Available at `www.systemc.org`.

[20] TARDITI, D., MORRISETT, G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, May 1996), ACM Press, New York, New York, pp. 181–192.

[21] WU, P., MIDKIFF, S., MOREIRA, J., AND GUPTA, M. Efficient support for complex numbers in Java. In *ACM Java Grande Conference* (San Fransisco, CA, 1999).

[22] YELICK, K., ET AL. Titanium: A high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing* (Stanford, California, Feb. 1998).

[23] YOUNG, J. S., MACDONALD, J., SHILMAN, M., TABBARA, A., HILFINGER, P., , AND NEWTON, A. R. Design and specification of embedded systems in Java using successive, formal refinement. In *Proceedings of the Design Automation Conference* (1998).